Polymorphism and Type Inference in Database Programming

Polymorphism and Type Inference in Database Programming

PETER BUNEMAN
University of Pennsylvania
and
ATSUSHI OHORI
Kyoto University

In order to find a static type system that adequately supports database languages, we need to express the most general type of a program that involves database operations. This can be achieved through an extension to the type system of ML that captures the polymorphic nature of field selection, together with a technique that generalizes relational operators to arbitrary data structures. The combination provides a statically typed language in which generalized relational databases may be cleanly represented as typed structures. As in ML types are inferred, which relieves the programmer of making the type assertions that may be required in a complex database environment.

These extensions may also be used to provide static polymorphic typechecking in object-oriented languages and databases. A problem that arises with object-oriented databases is the apparent need for dynamic typechecking when dealing with queries on heterogeneous collections of objects. An extension of the type system needed for generalized relational operations can also be used for manipulating collections of dynamically typed values in a statically typed language. A prototype language based on these ideas has been implemented. While it lacks a proper treatment of persistent data, it demonstrates that a wide variety of database structures can be cleanly represented in a polymorphic programming language.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classification—*applicative languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and construct, abstract data types*; H.2.1 [**Database Management**]: Logical Design—*data models and schemas*; H.2.3 [**Database Management**]: Languages—*database programming languages, query languages*

General Terms: Database Programming Languages, Type Systems, Data Models

---

This is the authors' version of the article published in ACM Transactions on Database Systems, 21(1):30-76, 1996.
Peter Buneman was partly supported by research grants NSF IRI86-10617 and ARO DAA6-29-84-K-0061; Atsushi Ohori's work was supported by Oki Electric Industry, Co., and by a Royal Society Research Fellowship at the University of Glasgow, Scotland.
Author's addresses: Peter Buneman: Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, U.S.A.; Atsushi Ohori: Research Institute for Mathematical Sciences, Kyoto University, Sakyo-ku, Kyoto 606-01, JAPAN.

Additional Key Words and Phrases: Generalized relational algebra, inheritance, object-oriented databases, polymorphism, record calculus, type inference

1. INTRODUCTION

Expressions such as 3 + "cat" and [Name = "J. Doe"].PartNumber contain *type errors* — applications of primitive operations such as "+" or "." (field selection) to inappropriate values. Static type checking – the detection of type errors in a program before it is executed – has long been advocated for many forms of database programming [Schmidt 1977; Albano et al. 1985; Atkinson and Buneman 1987; Kim 1994] which is characterized by the complexity and size of the data structures involved. For relational query languages checking of the type correctness of a query such as

```
select Name
from Employee
where Salary > 100000
```

is routinely carried out by the compiler, not only as a partial check on the correctness of the program, but also as an intrinsic part of the optimization process. Typechecking is straightforward because the type of the Employee relation is known from the database schema definitions. Suppose, however, these definitions were unavailable. The query itself still provides some type information: Employee is a relation, with at least an attribute Name of undetermined type and a numeric attribute Salary. How can we express the type information implied by the query? An equivalent problem is to express the most general type of the function

```
function Wealthy(S) = select Name
                      from S
                      where Salary > 100000
```

Here, the parameter S is constrained to be a relation with a Name and a numeric Salary attribute. However, describing this general constraint is not possible in current programming languages. In statically typed languages, one gives a complete type such as function Wealthy(S:EmployeeRel) $\cdots$ to the parameter of Wealthy, restricting the function to be applicable only to a particular relation type; and in dynamically typed languages, no type checking is done during compilation, allowing the possibility of run-time errors. The language ML has a *polymorphic* type system in which the most general type of a function can be described and inferred from its definition. However, ML's type system does not extend to database operations and cannot be used to describe the type of functions such as Wealthy. The purpose of this paper is to show how to extend the polymorphic type system of ML to database operations, and to demonstrate that the extended type system provides a practical basis for database programming languages where relational and object-oriented databases can be cleanly represented.

Why is such a polymorphic type system important to database programming, where one is generally working against a known schema, i.e. a fixed set of types? We see a number of reasons:

Separate Compilation/External Procedures. It is frequently advantageous to develop software components of large systems independently. One would like —

as far as possible — to check the type correctness of these components separately. This is the rationale for the development of systems of modules/packages for a number of programming languages [Ichbiah et al. 1979; Wirth 1977; Appel and MacQueen 1991]. In a database context one may well want to develop software independently of the schema or type definitions that consitute the database. Having a polymorphic type for a function such as Wealthy describes precisely the constraints placed on the schema by the query in the body of that function. Thus the type of Wealthy describes what must be checked when that code is linked to the database.

Schema Evolution. A common problem in database software maintenance is the need to cope with changes in the database schema. It is advantageous to be able to describe the precise constraints that the existing software places on the schema in order to describe what evolution is possible, or to identify the code that will have to be rewritten to cope with a given change of schema. Again, the polymorphic type of Wealthy describes precisely the constraint that the enclosed query places on the database schema.

Database Programming Languages. There is a growing interest in database programming languages with more expressive type systems. Traditional approaches, including object-oriented languages, have derived their type systems from programming languages and are either static [Schmidt 1977; Albano et al. 1985; Object Design Inc. 1991] or have some dynamic components [Atkinson et al. 1983; Copeland and Maier 1984]. However, neither of these approaches provides a satisfactory account of the polymorphic nature of database programming such as Wealthy above. Napier [Morrison et al. 1989] attempts to combine parametric polymorphism and persistence, but its polymorphism does not extend to operations on database structures. See [Atkinson and Buneman 1987] for a survey of various approaches to type-checking in database programming.

Type Inference. Finally there is an important point of programming convenience. ML has a *type inference* algorithm that automatically infers, from an untyped program, the most general (polymorphic) type of that program. This provides the programmer much of flexibility and convinience of dynamically typed languages such as Lisp while maintaining the type safety and some of the efficiency of statically typed languages. We believe it is highly desirable to extend these ideas to a database context, which is characterized by the complexity of the types involved.

This paper describes a prototype language Machiavelli developed at University of Pennsylvania, which embodies these ideas. A preliminary sketch of the language was presented in [Ohori et al. 1989]. In addition to polymorphic type system, Machiavelli also generalizes relational operators including join and projection to arbitrary complex objects, and contains a mechanism to represent statically typed programs over heterogeneous collections. In this paper, we shall describe Machiavelli, the principles on which its type system is constructed, its type inference algorithm to implement the type system, and demonstrate its expressive power in database programming. Machiavelli is implemented in Standard ML of New Jersey [Appel and MacQueen 1991] as an interpreter, that demonstrates the material presented here with the exception of reference types and cyclic data. Some related systems are worth mentioning:

SML♯, developed by one of the authors, is an extension of Standard ML [Milner et al. 1990] with the polymorphic record operations used in Machiavelli. Its compiler is implemented by modifying Standard ML of New Jersey compiler using an efficient compilation method for polymorphic record operations developed in [Ohori 1992; Ohori 1995]. *CAML light* has been extended by Rémy to include similar record operations based on his formulation of polymorphic typing for records [Remy 1989]. *CPL/Kleisli* is a system developed by Wong [Wong 1994] for heterogeneous database integration using the principles of programming with collection types[Buneman et al. 1994]. Its type system is derived from the one presented here and is an integral part of the query rewriting system that attempts to reformulate programs to exploit the optimizations that are available in the query languages associated with the external data sources. This system is in use by scientists involved with biomedical (genomic) databases [Hart and Wong 1994].

To illustrate a program in Machiavelli, consider the function Wealthy, which takes a set of records (i.e. a relation) with Name and Salary information and returns the set of all Name values that occur in records with Salary values over 100K. For example, applied to the relation

```
{[Name = "Joe", Salary = 22340],
 [Name = "Fred", Salary = 123456],
 [Name = "Helen", Salary = 132000]}
```

which is Machiavelli syntax for a set of records, we should get the set {"Fred", "Helen"} of character strings. The function is written in Machiavelli (whose syntax largely follows that of ML) as follows

```
fun Wealthy(S) = select x.Name
                 from x <- S
                 where x.Salary > 100000;
```

The select ... from ... where ... form is simple syntactic sugar for more basic Machiavelli program structure (see section 2).

Although no types are mentioned in the code, Machiavelli *infers* the type information

$$\mathsf{Wealthy} : \{d :: [\![\mathit{Name} : d', \mathit{Salary} : \mathit{int}]\!]\} \to \{d'\}.$$

In this type, d and d' are *type variables*, and the valid types for Wealthy may be obtained by substituting specific types for d and d'. However there are two restrictions on the types that may be substituted. The first is indicated by the decoration "$:: [\![\mathit{Name} : d', \mathit{Salary} : \mathit{int}]\!]$" on the type variable d. This allows only certain record types to be substituted for d, i.e. those with an integer *Salary* field, a *Name* field whose type must agree with the type in the output, and possibly other fields. Thus

$$\{[\mathit{Name} : \mathit{string}, \mathit{Salary} : \mathit{int}]\} \to \{\mathit{string}\}$$
$$\{[\mathit{Name} : \mathit{string}, \mathit{Age} : \mathit{int}, \mathit{Salary} : \mathit{int}]\} \to \{\mathit{string}\}$$
$$\{[\mathit{Name} : [\mathit{First} : \mathit{string}, \mathit{Last} : \mathit{string}], \mathit{Weight} : \mathit{int}, \mathit{Salary} : \mathit{int}]\} \to \{[\mathit{First} : \mathit{string}, \mathit{Last} : \mathit{string}]\}$$

are allowable instances of the type of Wealthy, while

$$\{[\mathit{Name} : \mathit{string}]\} \to \{\mathit{string}\}$$
$$\{[\mathit{Name} : \mathit{string}, \mathit{Age} : \mathit{int}, \mathit{Salary} : \mathit{string}]\} \to \{\mathit{string}\}$$
$$\{\mathit{int}\} \to \{\mathit{string}\}$$

are not allowable instances, for the substitutions for d that generate them do not match with the constraints imposed by the decoration $[\![Name : d', Salary : int]\!]$. Type variables whose instantiation is controlled by such a decoration are called *kinded* type variables.

The second constraint we place on the type variables d and d' is that they can only be instantiated with *description types*. Basic operations of databases require computable equality, and this is not available on function types and, may be unavailable on certain base types. Description types are essentially the same as ML's equality types – those types on which equality is available – but more operations are available. The constraint, whose details will be given later, governs how function types may appear in description types.

In order to display type variables using conventional programming fonts we follow the ML convention of displaying ordinary type variables as 'a, 'b, ... and description type variables as ''a, ''b etc. Thus the type $\{d :: [\![Name : d', Salary : int]\!]\} \to \{d'\}$ will be displayed in examples as {''a::[Name : ''b, Salary : int]} -> {''b}.

The typing Wealthy : {''a::[Name : ''b, Salary : int]} -> {''b} places restrictions on how Wealthy may be used. For example, all of the following will be rejected by the compiler.

```
Wealthy({[Name = "Joe"], [Name = "Fred"]})
Wealthy({[Name = "Joe", Salary = "nonsense"]})
sum(Wealthy({[Name = "Fred", Salary = 30000],
             [Name = "Joe", Salary = 200000]}))
```

In the first application the Salary field is missing; in the second it has the wrong type. In neither case can we find a suitable instantiation for the kinded type variable ''a::[Name : ''b, Salary : int]. In the third case we can find such an instantiation, but this results in the variable ''b being bound to string, so that the result of Wealthy is of type {string} — an inappropriate argument for sum.

There is a close relationship between the polymorphism represented by the kinded type variables and the generic nature of object-oriented programming. The type scheme {''a::[Name : ''b, Salary : int]} can be thought of as a class, and functions that are polymorphic with respect to this, such as Wealthy, can be thought of as methods of that class. For the purposes of finding a typed approach to object-oriented programming, Machiavelli's type system has similar goals to the systems proposed by Cardelli and Wegner [Cardelli 1988; Cardelli and Wegner 1985]. However, there are important technical differences, the most important of which is that Machiavelli does not use subtyping but uses polymorphic instantiation of kinded type variables to represent inheritance. This property allows us to capture the exact polymorphic nature of operations on records, and enables us to extend the type system to various database operations such as *natural join*.

Turning to object-oriented databases, research has centered more on a discussion of features[Atkinson et al. 1989] than on any principled attempt to provide a formal semantics. However, looking at these features, there are some that are not directly captured in a functional language with the relational extensions we have described above. First, the class structure of object-oriented languages provides a form of abstraction and inheritance that does not immediately fall out of an ML-style type system. Second, object identity is not provided in the relational model (though it is an open issue as to whether it requires more than the addition of a reference type, as

in ML.) Third, and perhaps most interesting from the standpoint of object-oriented databases, there is an implicit requirement that *heterogeneous* collections should be representable in the language. We believe that these issues can be satisfactorily resolved in the context of the type system we are advocating. In particular, we shall show how heterogeneous collections – which would appear to be inconsistent with static type-checking – can be satisfactorily represented using essentially the same apparatus developed to handle relational data types.

The organization of this paper is as follows. Section 2 introduces the basic data structures of Machiavelli including records, variants and sets, and shows how relational queries can be obtained with the operations for these structures. Section 3 contains a definition of the core language itself. It defines the syntax of types and terms, and describes the type inference system. Section 3 also presents the type inference process in some detail for the basic operations required for records, sets and variants. In section 4, the language is extended with relational operations – specifically join and projection – that cannot be derived from basic set operations, and the type inference system is extended to handle them. In section 5 we discuss how this type system can be used to capture an important aspect of object oriented databases, the manipulation of heterogeneous collections. Section 6 concludes with a brief discussion of further applications of these ideas to object-oriented languages and databases.

2. BASIC STRUCTURES FOR DATA REPRESENTATION

Our language and type system should be expressive enough to represent various data structures that violate the "first-normal-form assumption" underlying most implemented relational database systems and most of the traditional theory of relational databases. For example we want to be able to deal with structures such as

```
{[Name = [First = "Ellen", Last = "Gurman"], Children = {"Jeremy", "Christopher"}],
 [Name = [First = "Bridget", Last = "Ludford"], Children = {"Adam", "Benjamin"}]}
```

which is built up out of records and (uniformly typed) sets. This structure is a non-first-normal-form relation in which the Name field contains a record and the Children field contains a set of strings. It is an example of a *description term*, and in this section we shall describe the constructors that enable us to build up such terms from atomic data: records, variants, sets and references. We shall also describe how cyclic structures are created. As we describe each constructor, we shall say under what conditions it constructs a description term.

We start with the basic syntactic forms of Machiavelli for value and function definition, which are exactly those of ML. Names are bound to values by the use of val, as in

```
val four = 2 + 2
```

functions are defined through the use of fun, as in

```
fun factorial(n) = if eq(n,1) then 1 else n * factorial(n-1)
```

and there is a function constructor fn x => ... that is used to create functions without naming them, as in

```
(fn x => x + x) (4)
```

which evaluates to 8. There is also the form let $x = e_1$ in e_2 end, which evaluates e_2 in the environment in which x is bound to e_1. Example:

```
let x = 4 + 5 in x + x*x end
```

which evaluates to 90. This form is treated specially, and it is the basis for ML's polymorphism. By implicit or explicit use of let, polymorphic functions are introduced and used. A polymorphic function definition such as that of Wealthy is treated as shorthand for a let binding whose scope is the rest of the program.

2.1 Labeled Records and Labeled Variants

The syntax for labeled records is:

$[l_1 = v_1,\ldots,l_n = v_n]$

where $l_1,\ldots,l_n$ stand for *labels*. The labels in a record must be distinct and the order of their appearance is insignificant. A record is a description term if all its fields $v_1,\ldots,v_n$ are description terms. Other than record construction, ([...]), there are two primitives for records. The first, $\_.l$, is field selection; $r.l$ selects the l field from the record r. The second, modify(_,l,_), is field modification in which modify(r,l,e) creates a new record identical to r except on the l field where its value is e. For example,

```
modify([Name = "J. Doe", Age = 21], Age, 22)
```

evaluates to [Name = "J. Doe", Age = 22]. It is important to note that modify does *not* have a side-effect. It is a function that returns another record. This construct enables us to modify a record field that is not a reference and provides added flexibility in programming with records.

We shall make frequent use of the syntax (e_1, e_2) for pairs. This is simply an abbreviation for the record [first = e_1, second = e_2]. Triples and, generally, n-tuples are similarly constructed.

Variants are used to "tag" values in order to treat them uniformly. For example, the values <Int = 7> and <Real = 3.0> could both be treated as numbers, and the tags used to indicate how the value is to be interpreted (e.g. real or integer.) A program may use these tags in deciding what operations to perform on the tagged values (e.g. real or integer arithmetic.) The syntax for constructing a variant is:

<l=v>

The operation for analyzing a variant is a case expression:

```
case e of
    <l1=x1> => e1,
      ⋮
    <ln=xn> => en,
    else       e0
endcase
```

where each x_i in <l_i=x_i> => e_i is a variable whose scope is in e_i. This operation first evaluates e and if it yields a variant <l_i=v> then binds the variable x_i to the value v and evaluates e_i under this binding. The possible results $e_1,\ldots,e_n,e_0$ should all have the same type. If there is no matching case then the else clause is selected. The else is optional, and, if omitted, the argument e must be evaluated to a variant

labeled with one of $l_1, \ldots, l_n$. The type system ensures that this condition can be statically checked.

For example,

```
case <Consultant = [Name = "J. Doe", Address = "10 Main St.",
                    Phone = "222-1234"]>
of
   <Consultant = x> => x.Phone,
   <Employee = y> => y.Extension
endcase
```

yields "222-1234". Note that case ... endcase is an expression, and returns a value. A variant <$l = v$> is a description term if v is a description term.

2.2 Sets

Sets in Machiavelli can only contain description terms and sets themselves are always description terms. This restriction is essential to generalize database operations over structures containing sets. There are four basic expressions for sets:

$\{\}$	*empty set,*
$\{x\}$	*singleton set constructor,*
union(s_1,s_2)	*set union,*
hom(f,op,z,s)	*homomorphic extension.*

We use the notation $\{x_1, x_2, \ldots, x_n\}$ as shorthand for union($\{x_1\}$, union($\{x_2\}$, union(...,$\{x_n\}$))).

Of these, hom requires some explanation. This is a primitive function in Machiavelli, similar to the "pump" operation in FAD [Bancilhon et al. 1988] and the "fold" or "reduce" of many functional languages defined by

hom(f,op,z,$\{\}$) = z,
hom(f,op,z,$\{e\}$) = $f(e)$,
hom(f,op,z,union(e_1,e_2)) = op(hom(f,op,z,e_1),hom(f,op,z,e_2)).

For example, a function to check if there is at least one element satisfying property P in a set can be defined as

```
fun exists P S = hom(P, or, false, S)
```

and a function that finds the largest member of a set of non-negative integers is

```
fun max S = hom( fn x => x, fn(x,y) => if x > y then x else y, 0, S)
```

In general the result of this operation will depend on the order in which the elements of the set are encountered; however if *op* is an associative, commutative and idempotent operation with identity z and f has no side-effects (as is the case in the exists and max examples) then the result of hom will be independent of the order of this evaluation. Now one would also like to use hom on operations that are not idempotent, for example

```
fun sum S = hom(fn x => x, +, 0, S)
```

However + is not idempotent, and it is easy to construct programs whose outcome depends on the evaluation strategy [Breazu-Tannen and Subrahmanyam 1991]. It is easy enough to remove such ambiguous outcomes by insisting — as we have done in our implementation — that, in the representation of sets, we do not have duplicated

elements. This is equivalent to putting a condition on the third line of the definition of hom that the expressions e_1 and e_2 denote disjoint sets. Unfortunately this considerably complicates the operational semantics of the language, and it precludes the possibility of lazy evaluation. For a resolution of this issue see [Breazu-Tannen and Subrahmanyam 1991; Breazu-Tannen et al. 1991], which discuss the semantic properties of programs with sets and other collection types. In this paper we shall occasionally make use of "incorrect" applications of hom; however we are confident that the adoption of an alternative semantics will not affect typing issues, which are the main concern here.

Various useful functions can be defined using correct applications of hom. A function map(f, S), which applies the function f to each member of S is:

```
fun map(f,S) = hom(fn x => {f x}, union, {}, S)
```

For example map(max,{{1,2},{3},{6,5,4}}) evaluates to {2,3,6}. A selection function is defined by

```
fun filter(p,S) = hom(fn x => if p(x) then {x} else {}, union,{},S)
```

which extracts those members of S that satisfy property p; for example the expression filter(even,{1,2,3,4}) evaluates to {2,4}. In addition to these examples, hom can be used to define set intersection, membership in a set, set difference, the n-fold cartesian product (denoted by prod_n below) of sets and the powerset (the set of subsets) of a set. Also, the form

```
select E
from x1 <- S1,
     x2 <- S2,
       :
     xn <- Sn
where P
```

in which $x_1, x_2, \ldots, x_n$ may occur free in E and P, is provided in the spirit of relational query languages and the list comprehensions of Miranda [Turner 1985]. This can be implemented as

```
map((fn(e,p) => e),
  filter((fn(e,p) => p),
    map((fn(x1,x2,...,xn) => (E,P)),
      prod_n(S1,S2,...,Sn))))
```

See [Wadler 1990] for a related discussion of syntax for programming with lists.

2.3 Cyclic Structures

In many languages, the ability to define cyclic structures depends on the ability to reassign a pointer. In Machiavelli, these two ideas are separated. It is possible to create a structure with cycles through use of the (rec $v.e$) construct, e.g.

```
val Montana = (rec v.[Name = "Montana", Motto = "Big Sky Country",
                      Capital = [Name = "Helena", State = v]])
```

This record behaves like an infinite tree obtained by arbitrary unfolding by substitution for v. For example, the expressions Montana.Capital, Montana.Capital.State, Montana.Capital.State.Capital, etc. are all valid. Moreover, equality test and other

database operations on description terms generalize to those cyclic structures. This uniform treatment is achieved by treating description terms as *regular trees* [Courcelle 1983]. The syntax (rec $v.e$) denotes the regular tree given as the solution to the equation $v = e$, where e may contain the variable v. To ensure that the equation $v = e$ has a proper solution, we place the restriction that e must contain a proper term constructor (other than variables or (rec $v._-$) form).

2.4 References

We believe – though we shall comment more on this in section 6 – that the notion of object identity in databases is equivalent to that of references as they are implemented in ML. There are three primitives for references:

new(v)	*reference creation,*
!r	*de-referencing,*
r:=v	*assignment.*

new(v) creates a new reference and assigns the value v to it, !r returns the value associated with the reference r, and r:=v changes the value associated with the reference r to v. In a database context, they correspond respectively to creating an object with identity, retrieving the value of an object, and changing the associated value of an object without affecting its identity.

The uniqueness of identity is guaranteed by the uniqueness of each reference. Two references are equal only if they are the results of the same invocation of new primitive. For example if we create the following two *objects* (i.e. references to records),

```
John1 = new([Name="John", Age= 21]);
John2 = new([Name="John", Age= 21]);
```

then John1 = John1 and !John1 = !John2 are true but John1 = John2 is false even though their associated values are the same. Sharing and mutability are captured by references. If we define a department object as

```
SalesDept = new([Name = "Sales", Building = 11]);
```

and from this we define two employee objects as

```
John = new([Name="John", Age =21, Dept = SalesDept]);
Mary = new([Name="Mary", Age =31, Dept = SalesDept]);
```

then John and Mary *share* the same object SalesDept as the value of Dept field. Thus, an update to the object SalesDept as seen from John,

```
(!John).Dept := modify(!((!John).Dept), Building, 98)
```

is reflected in the department as seen from Mary. After this statement,

```
(!((!Mary).Dept)).Building
```

evaluates to 98. Unlike many languages references do not have an optional "nil" or "undefined" value. If such an option is required it must be explicitly introduced through the use of a variant. References are always description terms regardless of their associated values.

3. TYPE INFERENCE AND POLYMORPHISM IN MACHIAVELLI

Type inference is a technique used to infer type information that represents the polymorphic nature of a given untyped (or partially typed) program. Hindley [Hindley 1969] established a complete type inference algorithm for untyped lambda expressions. Independently, Milner [Milner 1978] developed a type inference algorithm for a functional programming language that allowed polymorphic definitions through use of the let construct. Damas and Milner [Damas and Milner 1982] later showed the completeness of this type inference algorithm, which has been successfully used in ML and other functional programming languages [Augustsson 1984; Milner et al. 1990; Turner 1985; Hudak et al. 1992]. In this section we give an account of the Damas-Milner type system and its extension which is used in Machiavelli to infer types for programs involving records, variants, and sets.

For programs that do not involve field selection, variants and database operations, Machiavelli infers type information similar to that of ML. For example, from the definition of the identity function

```
fun id x = x;
```

the type system infers the type

```
id : 'a -> 'a
```

where the type variable 'a stands for an "arbitrary type", and the notation 'a -> 'a represents the set of types that can be obtained by substituting 'a with various ground types (i.e. type expressions that do not contain type variables.) The most important property of the ML type system is that for any type consistent expression, a *principal type* can be inferred. This is a type whose instances are types of the expression and conversely any type of the expression is an instance.

A more substantial example of type inference is given by the function map of section 2.2, which has the type.

```
map : (("a -> "b) * {"a}) -> {"b}
```

Here "a and "b are type variables that represent description types. The type for map indicates that it is a function that takes a function of type $\delta_1 \to \delta_2$ and a set of type $\{\delta_1\}$ and returns a set of type $\{\delta_2\}$ where δ_1, δ_2 can be any description types. Thus map(max, {{1,2,3},{7},{5,2}}) is a legitimate application of map. Again, the type (("a -> "b) * {"a}) -> {"b} is principal in that any type for map is obtained by substituting description types for the type variables "a and "b. In the example, (({int} -> int) * {{int}}) -> {int} is the type of map in map(max, {{1,2,3},...}).

It is, however, not possible for ML's type inference method to infer a type for a program involving field selection, variants, or the operations of relational algebra that we shall describe later. In ML, the simplest function using field selection fun name x = x.Name requires an explicit type to be added for the argument x. The difficulty is that the type system of ML is not general enough to represent the relationship between a record type and the type of one of its fields.

Wand attempted [Wand 1987] to solve this problem using the notion of *row variables*, which are variables ranging over finite sets of record fields. His system, however, does not share with ML the property of principal typing (see [Ohori and Buneman 1988; Jategaonkar and Mitchell 1988; Wand 1988; Remy 1989] for analysis of the problem and for refinements.) Based on Wand's general observation, in

```
-> val joe = [Name="Joe", Age=21,
              Status=<Consultant = [Address="Philadelphia", Telephone=2221234]>];
>> val joe = [Name="Joe", Age=21,
              Status=<Consultant = [Address="Philadelphia", Telephone=2221234]>]
       : [Name : string, Age : int, Status : 'a::<Consultant : [Address : string,Telephone : int]>]
-> fun phone(x) = case x.Status of
                          <Employee = y> => y.Extension,
                          <Consultant = y> => y.Telephone
                  endcase
>> val phone = fn : 'a::[Status : <Employee : 'b::[Extension : 'd],
                                  Consultant : 'c::[Telephone : 'd]>] -> 'd
-> phone(joe);
>> val it = 2221234 : int
-> fun increment_age(x) = modify(x, Age, x.Age + 1);
>> val increment_age = fn : 'a::[Age : int] -> 'a::[Age : int]
-> increment_age([Name="John",Age=21]);
>> val it = [Name="John",Age=22] : [Name : string,Age : int]
```

Fig. 1. Some Simple Machiavelli Examples

[Ohori and Buneman 1988] a type inference method was developed that overcomes the difficulty and extends the method to database operations through the use of syntactic conditions to control substitution of type variables. The type system was further refined in [Ohori 1992; Ohori 1995] using kinded typing, which allows us to represent principal types of polymorphic recod operations. Machiavelli's type system is based on this account of record operations. For example, the function name above is given the type

```
name : 'a::[Name : 'b] -> 'b
```

The notation 'a::[Name : 'b] describes all record types containing the field Name : τ where τ is any instance of 'b. Substitutions for 'a are restricted to those that conform to this description. The type above then represents all possible types of the function name and may be taken as a principal (kinded) type for name. More examples of type inference for records and variants are shown in Figure 1 which shows an interactive session in Machiavelli. Input to the system is prompted by -> , and output is preceded by >> . The top level input is either a value or function binding; the variable it is a name for the result of evaluation of an expression. The output consists of this result together with its inferred type.

We now define a small language obtained by combining the data structures described in the previous section with a functional calculus and then giving its type system.

3.1 Expressions

The syntax of programs or *expressions* of the core language is given by

e ::= c_τ | () | x | (fn x => e) | $e(e)$ | let x=e in e end |
if e then e else e | eq(e,e) |
[l=e,...,l=e] | e.l | modify(e,l,e) |
<l=e> | case e of <l=x> => e,..., <l=x> => e endcase |
case e of <l=x> => e,..., <l=x> => e else => e endcase |
{} | {e} | union(e,e) | hom(e,e,e,e) |

$\mathsf{new}(e) \mid (!e) \mid e{:=}e \mid$
$(\mathsf{rec}\ x.e)$

In this, c_τ stands for standard constants including constants of base types and ordinary primitive functions on base types. x stands for the variables of the language. () is the single value of type unit and is returned by expressions such as assignment. The binding $\mathsf{val}\ id = e_1;\ e_2$ is syntactic sugar for $\mathsf{let}\ id = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$. Recursive function definition with multiple arguments can be defined using the standard method in functional languages such as ML. Evaluation rules for these expressions can be obtained by extending an operational semantics of ML (with references) such as that provided in [Tofte 1988].

3.2 Types and Description Types

The set of types of Machiavelli, ranged over by τ, is the set of regular trees [Courcelle 1983] represented by the following type expressions:

$$\tau ::= t \mid \mathsf{unit} \mid b \mid b_d \mid \tau \to \tau \mid [l{:}\tau,\ldots,l{:}\tau] \mid \langle l{:}\tau,\ldots,l{:}\tau\rangle \mid \{\tau\} \mid \mathsf{ref}(\tau) \mid (\mathsf{rec}\ v.\tau(v))$$

t stands for type variables. unit is the trivial type whose only value is (). b and b_d range respectively over the base types and base description types in the language. The other type expressions are: $\tau \to \tau$ for function types, $[l{:}\tau,\ldots,l{:}\tau]$ for record types, $\langle l{:}\tau,\ldots,l{:}\tau\rangle$ for variant types, and $\{\tau\}$ for set types. In $(\mathsf{rec}\ v.\tau(v))$, the body $\tau(v)$ is a type expression, in which the variable v may occur free, and the entire expression denotes the solution to the equation $v = \tau(v)$. To ensure that a type expression always denotes a unique regular tree, we place the restriction that $\tau(v)$ in $(\mathsf{rec}\ v.\tau(v))$ contains a proper type constructor (other than variables or $(\mathsf{rec}\ v'.\_)$ form). In keeping with our syntax for records we shall use the notation $\tau_1 * \tau_2$ as an abbreviation for the type $[\mathsf{first} : \tau_1, \mathsf{second} : \tau_2]$. n-tuple types are treated similarly.

Database examples of Machiavelli types are: a relation type,

```
{[PartNum : int, PartName : string, Color : <Red : unit, Green : unit, Blue : unit>]}
```

a complex object type,

```
{[Name : [First : string, Last : string], Children : {string}]}
```

and a mutable object type,

```
(rec p. ref([Id# : int, Name : string, Children : {p}]))
```

Note that $(\mathsf{rec}\ v.\tau(v))$ is not a type constructor but syntax to denote the solution to the equation $v = \tau(v)$. As a consequence, distinct type expressions may denote the same type. For example, the following type expression denotes the same type as the one above:

```
(rec p. ref([Id# : int, Name : string,
             Children : {ref([id# : int, Name : string, Children : {p}])}]))
```

The set of *description types*, ranged over by δ, is the subset of types represented by the following syntax:

$$\delta ::= d \mid \mathsf{unit} \mid b_d \mid [l{:}\delta,\ldots,l{:}\delta] \mid \langle l{:}\delta,\ldots,l{:}\delta\rangle \mid \{\delta\} \mid \mathsf{ref}(\tau) \mid (\mathsf{rec}\ v.\delta(v))$$

d stands for description type variables, i.e. those type variables whose instances are restricted to description types. τ in ref(τ) ranges over the syntax of all types given previously. This syntax forbids the use of a function type or a base type which is not a description type in a description type unless within a ref(...). Thus int -> int is not a description type but

```
ref([x_coord : int, y_coord : int, move_horizontal : int -> ()])
```

is a description type.

3.3 Type Inference without Records and Variants

A legal Machiavelli program corresponds to an (untyped) expression associated with a type inferred by the type inference system. As such, the definition of this implicit system requires two steps: first we give the *typing rules*, which determine when an untyped expression e is judged to have a type τ and is therefore well typed; second, we develop a type inference algorithm that infers, for any type consistent expression, a principal type. For readability, we develop the description of the type system, in two stages. In this and the following subsection, we describe the type system for expressions that do not involve records and variants; then, in subsection 3.4, we extend the system to records and variants by introducing kinding.

The typing rules are given as a set of rules to derive *typing judgments*. Since, in general, an expression e contains free variables and the type of e depends on the types assigned to those variables, a typing judgment is defined relative to a type assignment of free variables. We let $\mathcal{A}$ range over type assignments, which are functions from a finite subset of variables to types. We write $\mathcal{A}(x, \tau)$ for the function $\mathcal{A}'$ such that $domain(\mathcal{A}') = domain(\mathcal{A}) \cup \{x\}$, $\mathcal{A}'(x) = \tau$ and $\mathcal{A}'(y) = \mathcal{A}(y)$ for $y \neq x$. A typing judgment is a formula of the form:

$$\mathcal{A} \rhd e : \tau$$

expressing the fact that expression e has type τ under type assignment $\mathcal{A}$. The typing rules for those operations in Machiavelli that do not involve records are shown in Figure 2. Note that in some of them such as (UNION), types are restricted to description types, which is indicated by the use of δ instead of τ.

In (LET), the notation $e_1[e_2/x]$ denotes the expression obtained from e_1 by substituting e_2 for all free occurrences of x. This is a departure from the Damas-Milner system [Damas and Milner 1982] in that it does not use generic types (a type expression of the form $\forall t.\, \tau$) but instead it uses syntactic substitution of expressions. It can be shown [Ohori 1989a; Mitchell 1990] that this proof system is equivalent to that of Damas-Milner. The advantage of our treatment of let is that it yields simpler proofs and can be extended to the relational algebra, as we shall show later. While it is possible to extend Damas-Milner generic types to records and variants using kinded type abstraction [Ohori 1992; Ohori 1995], we do not know how to extend this technique to the conditional typing that we shall require for database operations such as join and projection. Note that this rule is only to define typing for let expressions; it does not imply that the semantics for a let expression is defined by term substitution, which would yield call-by-name semantics. Since we have references, we choose the usual call-by-value semantics for let expressions.

The proof system of Figure 2 determines which expressions are type correct Machiavelli programs (not involving operations on records and variants). Unlike

$$(\textsc{const}) \quad \mathcal{A} \rhd c_\tau \ : \ \tau$$

$$(\textsc{unit}) \quad \mathcal{A} \rhd () \ : \ \mathsf{unit}$$

$$(\textsc{var}) \quad \mathcal{A} \rhd x \ : \ \tau \quad \text{if } x \in \mathit{domain}(\mathcal{A}), \mathcal{A}(x) = \tau$$

$$(\textsc{abs}) \quad \frac{\mathcal{A}(x, \tau_1) \rhd e \ : \ \tau_2}{\mathcal{A} \rhd \mathsf{fn}\ x\ \mathsf{=>}\ e \ : \ \tau_1 \to \tau_2}$$

$$(\textsc{app}) \quad \frac{\mathcal{A} \rhd e_1 \ : \ \tau_1 \to \tau_2 \qquad \mathcal{A} \rhd e_2 \ : \ \tau_1}{\mathcal{A} \rhd e_1(e_2) \ : \ \tau_2}$$

$$(\textsc{let}) \quad \frac{\mathcal{A} \rhd e_1[e_2/x] \ : \ \tau \qquad \mathcal{A} \rhd e_2 \ : \ \tau'}{\mathcal{A} \rhd \mathsf{let}\ x = e_2\ \mathsf{in}\ e_1\ \mathsf{end} \ : \ \tau}$$

$$(\textsc{if}) \quad \frac{\mathcal{A} \rhd e_1 \ : \ \mathsf{bool} \qquad \mathcal{A} \rhd e_2 \ : \ \tau \qquad \mathcal{A} \rhd e_3 \ : \ \tau}{\mathcal{A} \rhd \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \ : \ \tau}$$

$$(\textsc{eq}) \quad \frac{\mathcal{A} \rhd e_1 \ : \ \delta \qquad \mathcal{A} \rhd e_2 \ : \ \delta}{\mathcal{A} \rhd \mathsf{eq}(e_1,e_2) \ : \ \mathsf{bool}}$$

$$(\textsc{singleton}) \quad \frac{\mathcal{A} \rhd e \ : \ \delta}{\mathcal{A} \rhd \{e\} \ : \ \{\delta\}}$$

$$(\textsc{union}) \quad \frac{\mathcal{A} \rhd e_1 \ : \ \{\delta\} \qquad \mathcal{A} \rhd e_2 \ : \ \{\delta\}}{\mathcal{A} \rhd \mathsf{union}(e_1,e_2) \ : \ \{\delta\}}$$

$$(\textsc{hom}) \quad \frac{\mathcal{A} \rhd e_1 \ : \ \delta \to \tau_1 \quad \mathcal{A} \rhd e_2 \ : \ (\tau_1 * \tau_2) \to \tau_2 \quad \mathcal{A} \rhd e_3 \ : \ \tau_2 \quad \mathcal{A} \rhd e_4 \ : \ \{\delta\}}{\mathcal{A} \rhd \mathsf{hom}(e_1,e_2,e_3,e_4) \ : \ \tau_2}$$

$$(\textsc{new}) \quad \frac{\mathcal{A} \rhd e \ : \ \tau}{\mathcal{A} \rhd \mathsf{new}(e) \ : \ \mathsf{ref}(\tau)}$$

$$(\textsc{deref}) \quad \frac{\mathcal{A} \rhd e \ : \ \mathsf{ref}(\tau)}{\mathcal{A} \rhd !e \ : \ \tau}$$

$$(\textsc{assign}) \quad \frac{\mathcal{A} \rhd e_1 \ : \ \mathsf{ref}(\tau) \qquad \mathcal{A} \rhd e_2 \ : \ \tau}{\mathcal{A} \rhd e_1{:=}e_2 \ : \ \mathsf{unit}}$$

$$(\textsc{rec}) \quad \frac{\mathcal{A}(v, \delta) \rhd e(v) \ : \ \delta}{\mathcal{A} \rhd (\mathsf{rec}\ v.\ e(v)) \ : \ \delta}$$

Fig. 2. Typing Rules for Expressions Without Records and Variants

the simple type discipline, this proof system does not immediately yield a decision procedure for type checking expressions. The second step of the definition of the type system is to give such a decision procedure. Following [Hindley 1969; Milner 1978], we solve this problem by developing an algorithm that always infers a principal typing for any type consistent expressions.

A *substitution* S is a function from type variables to types. A substitution may be extended to type expressions, and we identify a substitution and its extension, i.e. we shall write $S(\tau)$ for the expression obtained by replacing each type variable t in τ with $S(t)$. A typing $\mathcal{A}_1 \rhd e : \tau_1$ is *more general* than $\mathcal{A}_2 \rhd e : \tau_2$ if $domain(\mathcal{A}_1) \subseteq domain(\mathcal{A}_2)$ and there is some substitution S such that $\tau_2 = S(\tau_1)$ and $\mathcal{A}_2(x) = S(\mathcal{A}_1(x))$ for all $x \in domain(\mathcal{A}_1)$. A typing $\mathcal{A} \rhd e : \tau$ is *principal* if it is more general than any other derivable typing of e.

Figure 3 shows an algorithm to compute a principal typing for any untyped expression of Machiavelli that does not contain records, variants and database operations. The algorithm consists of a set of functions, one for each typing rule, together with the main function $Typing$. Based on the typing rule (RULE), P_{RULE} synthesizes a principal typing for an expression e from those of its subexpressions. It generates the equations that make the typings of the subexpressions conform to the premises of the rule, solves the equations and generates the typing corresponding to the conclusion of the rule. *Unify* used in these functions is a unification algorithm; $allpairs(\{\mathcal{A}_1, \ldots, \mathcal{A}_n\})$ denotes the set of pairs $\{(\mathcal{A}_i(x), \mathcal{A}_j(x)) \mid x \in domain(\mathcal{A}_i) \cap domain(\mathcal{A}_j), i \neq j\}$. The notation $F{\uparrow}^X$ denotes the restriction of the function F to the set $X \subseteq domain(F)$.

For example, consider the function P_{APP}, which takes principal typings of e_1 and e_2, and synthesizes a principal typing of $e_1(e_2)$. It first generates the equations that require the common variables of e_1 and e_2 to have the same type assignment, together with the equation that makes the type of e_2 to be the domain type of the type of e_1. They are respectively the set of equations $allpairs(\{\mathcal{A}_1, \mathcal{A}_2\})$ and the equation $(\tau_1, \tau_2 \rightarrow t)$. It then solves these equations by *Unify* which always finds a most general solution to the equations (if it exists) in the form of a substitution S. Finally, it returns the type assignment $S(\mathcal{A}_1) \cup S(\mathcal{A}_2)$ and a type $S(t)$, corresponding to the conclusion of the rule APP.

The main function $Typing$ is presented in the style of [Mitchell 1990]. It analyzes the structure of the given expression, recursively calls itself on its subexpressions to get their principal typings and then calls an appropriate function P that corresponds to the outermost constructor of the expression. The extra parameter L to $Typing$ is an environment that records the principal typings of let-bound variables. By maintaining this environment, the algorithm avoids repeated computation of a principal type of e_1 in inferring a typing of expressions of the form let x=e_1 in e_2 end, and it also enables incremental compilation. Renaming type variables in the case of $x \in domain(L)$ effectively achieves the same effect of computing the principal typing of e_1 for each occurrence of x in e_2.

As an example of type inference, let us use the algorithm to compute a principal typing of the function insert and of its application:

```
val insert = fn x => fn S => union({x}, S);
insert 2 {};
```

$P_{\text{APP}}((\mathcal{A}_1, \tau_1), (\mathcal{A}_2, \tau_2)) =$
  let $S = \mathit{Unify}(\mathit{allpairs}(\{\mathcal{A}_1, \mathcal{A}_2\}) \cup \{(\tau_1, \tau_2 \to t)\})$ (t fresh)
  in $(S(\mathcal{A}_1) \cup S(\mathcal{A}_2), S(t))$
  end

$P_{\text{ABS}}((\mathcal{A}, \tau), x) =$
  if $x \in \mathsf{domain}(\mathcal{A})$ *then* $(\mathcal{A}^{\uparrow \mathit{domain}(\mathcal{A}) \setminus \{x\}}, \mathcal{A}(x) \to \tau)$
  else $(\mathcal{A}, t \to \tau)$ (t fresh)

$P_{\text{LET}}((\mathcal{A}_1, \tau_1), (\mathcal{A}_2, \tau_2)) =$
  let $S = \mathit{Unify}(\mathit{allpairs}(\{\mathcal{A}_1, \mathcal{A}_2\}))$
  in $(S(\mathcal{A}_1 \cup \mathcal{A}_2), S(\tau_2))$
  end

$P_{\text{SINGLETON}}(\mathcal{A}, \tau) = \mathit{let}\ S = \mathit{Unify}(\{(\tau, d)\})\ \mathit{in}\ (S(\mathcal{A}), \{S(d)\})\ \mathit{end}$ (d fresh)

$P_{\text{UNION}}((\mathcal{A}_1, \tau_1), (\mathcal{A}_2, \tau_2)) =$
  let $S = \mathit{Unify}(\mathit{allpairs}(\{\mathcal{A}_1, \mathcal{A}_2\}) \cup \{(\tau_1, \tau_2), (\tau_1, \{d\})\})$ (d fresh)
  in $(S(\mathcal{A}_1 \cup \mathcal{A}_2), S(\{d\}))$
  end

⋮

$\mathit{Typing}(e, L) =$
  case e of:

c_τ	$\Longrightarrow$	$(\emptyset, \tau)$
x	$\Longrightarrow$	if $x \in \mathit{domain}(L)$ then $L(x)$ with all type variables renamed else $(\{x : t\}, t)$ (t fresh)
fn x => e	$\Longrightarrow$	$P_{\text{ABS}}(\mathit{Typing}(e, L), x)$
$e_1(e_2)$	$\Longrightarrow$	$P_{\text{APP}}(\mathit{Typing}(e_1, L), \mathit{Typing}(e_2, L))$
let $x = e_1$ in e_2	$\Longrightarrow$	let $(\mathcal{A}_1, \tau_1) = \mathit{Typing}(e_1, L)$ $L' = L(x, (\mathcal{A}_1, \tau_1))$ in $P_{\text{LET}}((\mathcal{A}_1, \tau_1), \mathit{Typing}(e_2, L'))$
$\{e\}$	$\Longrightarrow$	$P_{\text{SINGLETON}}(\mathit{Typing}(e, L))$
union(e_1,e_2)	$\Longrightarrow$	$P_{\text{UNION}}(\mathit{Typing}(e_1, L), \mathit{Typing}(e_2, L))$
⋮		

  endcase

Fig. 3. Type Inference Algorithm without Records, Variants

Figure 4 shows the sequence of the function calls and their results during the computation. Line 1 is the top level call of the algorithm on fn x => fn S => union({x}, S). Line 3 is the first recursive call on its only subexpression, whose result is shown on line 16. Lines 9 and 12 contain a call of *Typing* on a variable, which immediately returns a principal typing. In $P_{\text{SINGLETON}}$ on line 10 and 11, type variable t_1 is unified with a fresh description type variable d_1. In line 13 and 14, P_{UNION} unifies type variable t_2 with type $\{d_1\}$ and takes the union of type assignments. Line 18 shows a principal typing of insert. Lines 19 – 36 show the process for insert 2 {}, which is a shorthand for let insert = fn x => fn S => union({x}, S) in insert 2 {} end.

It requires some work to show that the algorithm we have described has the desired properties. We have also glossed over some important details such as the treatment of description type variables, recursive types and references. Before dealing with these issues let us first show how the typing rules and the inference system may be extended to handle records and variants.

3.4 Kinded Type Inference for Records and Variants

To extend the type system to records and variants, we need to introduce *kind constraints* on type variables. The set of kinds in Machiavelli is given by the syntax:

$$K ::= \mathsf{U} \mid [\![l{:}\tau,\ldots,l{:}\tau]\!] \mid \langle\!\langle l{:}\tau,\ldots,l{:}\tau\rangle\!\rangle$$

The idea is that U denotes the set of all types, $[\![l_1{:}\tau_1,\ldots,l_n{:}\tau_n]\!]$ denotes the set of record types containing the set of all fields $l_1 : \tau_1, \ldots, l_n : \tau_n$, and $\langle\!\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n\rangle\!\rangle$ denotes the set of variant types containing the set of all variants $l_1 : \tau_1, \ldots, l_n : \tau_n$.

In the extended type system, type variables must be kinded by a *kind assignment* $\mathcal{K}$, which is a mapping from type variables to kinds. We write $\{t_1 :: k_1, \ldots, t_n :: k_n\}$ for a kind assignment $\mathcal{K}$ that maps t_i to k_i $(1 \leq i \leq n)$. A type τ has a kind k under a kind assignment $\mathcal{K}$, denoted by $\mathcal{K} \vdash \tau :: k$, if it satisfies the conditions shown in Figure 5. For example, the following is a legal kinding:

$$\{t_1 :: \mathsf{U}, t_2 :: [\![\mathsf{Name} : t_1,\ \mathsf{Age} : \mathsf{int}]\!]\} \vdash t_2 :: [\![\mathsf{Name} : t_1]\!]$$

A typing judgment is now refined to incorporate kind constraints on type variables:

$$\mathcal{K}, \mathcal{A} \rhd e : \tau$$

Typing judgments of the form $\mathcal{A} \rhd e : \tau$ described in the previous subsection should now be taken as judgments of the form $\mathcal{K}_0, \mathcal{A} \rhd e : \tau$ where $\mathcal{K}_0$ is the kind assignment mapping all the type variables appearing in $\mathcal{A}, \tau$ to the universal kind U. The typing rules for records and variants in the extended type system are given in Figure 6. The rules for other constructors are the same as before except that they should be reinterpreted relative to a kind assignment $\mathcal{K}$. For example, the rule ABS becomes

$$(\text{ABS}) \qquad \frac{\mathcal{K}, \mathcal{A}(x, \tau_1) \rhd e : \tau_2}{\mathcal{K}, \mathcal{A} \rhd \mathsf{fn}\ x\ \mathsf{=>}\ e : \tau_1 \to \tau_2}$$

In particular, these propagate the kind assignment, but they do not change it, nor do they involve kinding judgements of the form $\mathcal{K} \vdash \tau :: K$.

It can be seen that the kinding constraints in the rules (DOT) and (VARIANT) express the conditions under which field selection and variant formation can be

$Typing$(fn x => fn S => union({x},S),$\emptyset$)
$= P_{\text{ABS}}(Typing(\text{fn S => union}(\{x\},S),\emptyset),x)$
$\rangle Typing(\text{fn S => union}(\{x\},S),\emptyset)$
$\rangle \quad = P_{\text{ABS}}(Typing(\text{union}(\{x\},S),\emptyset),S)$
$\rangle \quad \rangle Typing(\text{union}(\{x\},S),\emptyset)$
$\rangle \quad \rangle \quad = P_{\text{UNION}}(Typing(\{x\},\emptyset),Typing(S,\emptyset))$
$\rangle \quad \rangle \quad \rangle Typing(\{x\},\emptyset)$
$\rangle \quad \rangle \quad \rangle \quad = P_{\text{SINGLETON}}(Typing(x,\emptyset))$
$\rangle \quad \rangle \quad \rangle \quad \rangle Typing(x,\emptyset) = (\{x\colon t_1\},t_1)$
$\rangle \quad \rangle \quad \rangle \quad = P_{\text{SINGLETON}}((\{x\colon t_1\},t_1))$
$\rangle \quad \rangle \quad \rangle \quad = (\{x\colon d_1\},\{d_1\})$
$\rangle \quad \rangle \quad \rangle Typing(S,\emptyset) = (\{S\colon t_2\},t_2)$
$\rangle \quad \rangle \quad = P_{\text{UNION}}((\{x\colon d_1\},\{d_1\}),(\{S\colon t_2\},t_2))$
$\rangle \quad \rangle \quad = (\{x : d_1, S : \{d_1\}\},\{d_1\})$
$\rangle \quad = P_{\text{ABS}}((\{x : d_1, S : \{d_1\}\},\{d_1\}),S)$
$\rangle \quad = (\{x : d_1\},\{d_1\} \to \{d_1\})$
$= P_{\text{ABS}}((\{x : d_1\},\{d_1\} \to \{d_1\}),x)$
$= (\emptyset, d_1 \to \{d_1\} \to \{d_1\})$

$Typing$(let insert = fn x => fn S => union({x},S) in insert 2 {} end,$\emptyset$)
$= P_{\text{LET}}((\emptyset, d_1 \to \{d_1\} \to \{d_1\})), Typing(\text{insert 2 }\{\},\{(\text{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\})$
$\rangle Typing(\text{insert 2 }\{\},\{(\text{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\})$
$\rangle \quad = P_{\text{APP}}(Typing(\text{insert 2},\{(\text{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\}),$
$\qquad Typing(\{\},\{(\text{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\}))$
$\rangle \quad \rangle Typing(\text{insert 2},\{(\text{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\})$
$\rangle \quad \rangle \quad = P_{\text{APP}}(Typing(\text{insert},\{(\text{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\}),$
$\qquad Typing(2,\{(\text{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\}))$
$\rangle \quad \rangle \quad \rangle Typing(\text{insert},\{(\text{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\})$
$\rangle \quad \rangle \quad \rangle \quad = (\emptyset, d_2 \to \{d_2\} \to \{d_2\})$
$\rangle \quad \rangle \quad \rangle Typing(2,\{(\text{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\})$
$\rangle \quad \rangle \quad \rangle \quad = (\emptyset, int)$
$\rangle \quad \rangle \quad = P_{\text{APP}}((\emptyset, d_2 \to \{d_2\} \to \{d_2\}),(\emptyset, int))$
$\rangle \quad \rangle \quad = (\emptyset, \{int\} \to \{int\})$
$\rangle \quad \rangle Typing(\{\},\{(\text{insert},(\emptyset, d_1 \to \{d_1\} \to \{d_1\}))\})$
$\rangle \quad \rangle \quad = (\emptyset, \{d_3\})$
$\rangle \quad = P_{\text{APP}}((\emptyset, \{int\} \to \{int\}),(\emptyset, \{d_3\}))$
$\rangle \quad = (\emptyset, \{int\})$
$= P_{\text{LET}}((\emptyset, d_1 \to \{d_1\} \to \{d_1\}),(\emptyset, \{int\}))$
$= (\emptyset, \{int\})$

Fig. 4. Computing a Principal Typing

$$\mathcal{K} \vdash \tau :: \mathsf{U} \quad \text{for all } \tau$$
$$\mathcal{K} \vdash t :: [\![l_1{:}\tau_1,\ldots,l_n{:}\tau_n]\!] \quad \text{if } t \in \mathit{domain}(\mathcal{K}), \mathcal{K}(t) = [\![l_1{:}\tau_1,\ldots,l_n{:}\tau_n,\ldots]\!]$$
$$\mathcal{K} \vdash [l_1{:}\tau_1,\ldots,l_n{:}\tau_n,\ldots] :: [\![l_1{:}\tau_1,\ldots,l_n{:}\tau_n]\!]$$
$$\mathcal{K} \vdash t :: \langle\!\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n\rangle\!\rangle \quad \text{if } t \in \mathit{domain}(\mathcal{K}), \mathcal{K}(t) = \langle\!\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n,\ldots\rangle\!\rangle$$
$$\mathcal{K} \vdash \langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n,\ldots\rangle :: \langle\!\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n\rangle\!\rangle$$

Fig. 5. Kinding Rules

$$\text{(RECORD)} \quad \frac{\mathcal{K},\mathcal{A} \rhd e_1 : \tau_1, \ \ldots\ , \mathcal{K},\mathcal{A} \rhd e_n : \tau_n}{\mathcal{K},\mathcal{A} \rhd [l_1{=}e_1,\ldots,l_n{=}e_n] : [l_1{:}\tau_1,\ldots,l_n{:}\tau_n]}$$

$$\text{(DOT)} \quad \frac{\mathcal{K},\mathcal{A} \rhd e : \tau_1 \qquad \mathcal{K} \vdash \tau_1 :: [\![l : \tau_2]\!]}{\mathcal{K},\mathcal{A} \rhd e.l : \tau_2}$$

$$\text{(MODIFY)} \quad \frac{\mathcal{K},\mathcal{A} \rhd e_1 : \tau_1 \qquad \mathcal{K},\mathcal{A} \rhd e_2 : \tau_2 \qquad \mathcal{K} \vdash \tau_1 :: [\![l : \tau_2]\!]}{\mathcal{K},\mathcal{A} \rhd \mathsf{modify}(e_1,l,e_2) : \tau_1}$$

$$\text{(VARIANT)} \quad \frac{\mathcal{K},\mathcal{A} \rhd e : \tau_1 \qquad \mathcal{K} \vdash \tau_2 :: \langle\!\langle l{:}\tau_1\rangle\!\rangle}{\mathcal{K},\mathcal{A} \rhd \langle l{=}e\rangle : \tau_2}$$

$$\text{(CASE)} \quad \frac{\mathcal{K},\mathcal{A} \rhd e : \langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n\rangle \qquad \mathcal{K},\mathcal{A}(x_i,\tau_i) \rhd e_i : \tau \ (1 \le i \le n)}{\mathcal{K},\mathcal{A} \rhd \mathsf{case}\ e\ \mathsf{of}\ \langle l_1{=}x_1\rangle \Rightarrow\ e_1,\ \ldots,\ \langle l_n{=}x_n\rangle \Rightarrow\ e_n\ \mathsf{endcase} : \tau}$$

$$\text{(CASE')} \quad \frac{\begin{array}{c}\mathcal{K},\mathcal{A} \rhd e : \tau_0 \\ \mathcal{K},\mathcal{A}(x_i,\tau_i) \rhd e_i : \tau (1 \le i \le n) \\ \mathcal{K},\mathcal{A} \rhd e_0 : \tau \\ \mathcal{K} \vdash \tau_0 :: \langle\!\langle l_1{:}\tau_1,\ldots,l_n{:}\tau_n\rangle\!\rangle\end{array}}{\mathcal{K},\mathcal{A} \rhd\ \mathsf{case}\ e\ \mathsf{of}\ \langle l_1{=}x_1\rangle \Rightarrow\ e_1,\ldots,\ \langle l_n{=}x_n\rangle \Rightarrow\ e_n\ \mathsf{else} \Rightarrow\ e_0\ \mathsf{endcase} : \tau}$$

Fig. 6. Typing Rules for Records and Variants

typed. The following is an example of a legal typing:

$$\{t_1 :: \mathsf{U}, t_2 :: [\![\mathsf{Name} : t_1]\!]\}, \emptyset \rhd \mathsf{fn}\ \mathsf{x} \Rightarrow\ \mathsf{x.Name} : t_2 \to t_1$$

which says that the function fn x => x.Name can be applied to any record type t_2 which contains the field Name : t_1 and returns a value of type t_1.

Note that we do not need "recursive kinds" to represent recursive polymorphic types involving records and variants. They are represented by a kind assignment in which type variables may assigned to a kind containing some of those type variables, as seen in the following example:

$$\{d :: [\![\mathsf{Children}{:}\{d\}]\!]\}, \emptyset \rhd \mathsf{fn}\ \mathsf{x} \Rightarrow\ \mathsf{union}(\{\mathsf{x}\}, \mathsf{x.Children}) : d \to \{d\}$$

Because of the cyclic dependency of the kind constraint, instances of the description type variable d are restricted to the recursive types of the form (rec p.[Children:{p},...]). The following is an example of an instance of this type scheme:

$$(\mathsf{rec\ p.[Name{:}string,\ Children{:}\{p\}]}) \to \{(\mathsf{rec\ p.[Name{:}string,\ Children{:}\{p\}]})\}$$

To refine the type inference algorithm, we need to refine an unification algorithm to *kinded unification*. The strategy is to add a kind assignment to each component in unification and to check the condition that unification respects the constraints specified by kind assignments. A *kinded substitution* is a pair $(\mathcal{K}, S)$ consisting of a kind assignment $\mathcal{K}$ and a substitution S. Intuitively, the kind assignment $\mathcal{K}$ is the

kind constraints that must be satisfied by the results of applying the substitution S. We write $[t_1 \mapsto \tau_1, \ldots, t_n \mapsto \tau_n]$ for the substitution S such that $S(t_i) = \tau_i$ $(1 \leq i \leq n)$, and $S(x) = x$ for all the other variables. We say that a kinded substitution $(\mathcal{K}_1, S)$ *respects* a kind assignment $\mathcal{K}_2$ if, for all $t \in \mathit{domain}(\mathcal{K}_2)$, $\mathcal{K}_1 \vdash S(t) :: S(\mathcal{K}_2(t))$ is a legal kinding. For example, a kind substitution

$$(\{t_1 :: \mathsf{U}\}, [t_2 \mapsto [\mathsf{Name} : t_1, \mathsf{Age} : \mathsf{int}]])$$

respects the kind constraints $\{t_1 :: \mathsf{U}, t_2 :: [\![\mathsf{Name} : t_1]\!]\}$ and can be applied to type t_2 under this constraint. A kinded substitution $(\mathcal{K}_1, S_1)$ is *more general* than $(\mathcal{K}_2, S_2)$ if $S_2 = S_3 \circ S_1$ for some S_3 such that $(\mathcal{K}_2, S_3)$ respects $\mathcal{K}_1$, where $S \circ S'$ is the composition of substitutions S, S' defined as $S \circ S'(t) = S(S'(t))$. A *kinded set of equations* is a pair consisting of a kind assignment and a set of pairs of types. A kinded substitution $(\mathcal{K}_1, S)$ is a *unifier* of a kinded set of equations $(\mathcal{K}_2, E)$ if it respects $\mathcal{K}_2$ and $S(\tau_1) = S(\tau_2)$ for all $(\tau_1, \tau_2) \in E$. We can then obtain the following result, a refinement of Robinson's [Robinson 1965] unification algorithm.

THEOREM 1. *There is an algorithm Unify which, given any kinded set of equations, computes a most general unifier if one exists and reports failure otherwise.*

We provide here a description of the algorithm for acyclic types; its correctness proof can be found in [Ohori 1995]. We will describe the necessary extensions for cyclic regular trees in the following subsection.

The algorithm *Unify* is presented in the style of [Gallier and Snyder 1989] by a set of transformation rules on triples $(\mathcal{K}, E, S)$ consisting of a kind assignment $\mathcal{K}$, a set E of type equations and a set S of "solved" type equations of the form (t, τ) such that $t \notin \mathit{FTV}(\tau)$. Let $(\mathcal{K}, E)$ be a given kinded set of equations. The algorithm *Unify* first transforms $(\mathcal{K}, E, \emptyset)$ to $(\mathcal{K}', E', S')$ until no more rules can apply. It then returns $(\mathcal{K}', S')$ if E' is empty; otherwise it reports failure.

Let F range over functions from a finite set of labels to types. We write $[F]$ and $[\![F]\!]$ respectively to denote the record type identified by F and the record kind identified by F. Figure 7 gives the set of transformation rules for record types and function types. The rules for variant types are obtained from those of record types by replacing record type constructor $[F]$, record kind constructor $[\![F]\!]$ with variant type constructor $\langle F \rangle$, and variant kind constructor $\langle\!\langle F \rangle\!\rangle$, respectively. Rules I, II, V and VI are same as those in ordinary unification. Rule I eliminates an equation that is always valid. Rule II is the case for variable elimination; if occur-check (the condition that t does not appear in τ) succeeds then it generates one point substitution $[t \mapsto \tau]$, applies it to all the type expressions involved and then moves the equation (t, τ) to the solved position. Rules V and VI decompose an equation of complex types into a set of equations of the corresponding subcomponents. Rules III and IV are cases for variable elimination similar to rule II except that the variables have non trivial kind constraint. In addition to eliminating a type variable as in rule II, these rules check the consistency of kind constraints and, if they are consistent, generates a set of new equations equivalent to the kind constraints.

Using this refined unification algorithm, we can now extend the type inference system. First, we refine the notion of principal typings. A typing $\mathcal{K}_1, \mathcal{A}_1 \rhd e : \tau_1$ is *more general than* $\mathcal{K}_2, \mathcal{A}_2 \rhd e : \tau_2$ if $\mathit{domain}(\mathcal{A}_1) \subseteq \mathit{domain}(\mathcal{A}_2)$, and there is a substitution S such that the kinded substitution $(\mathcal{K}_2, S)$ respects $\mathcal{K}_1$, $\mathcal{A}_2(t) =$

I $(\mathcal{K}, E \cup \{(\tau, \tau)\}, S) \Rightarrow (\mathcal{K}, E, S)$

II $(\mathcal{K} \cup \{t \mapsto U\}, E \cup \{(t, \tau)\}, S) \Rightarrow ([t \mapsto \tau](\mathcal{K}), [t \mapsto \tau](E), \{(t, \tau)\} \cup [t \mapsto \tau](S))$
if t does not appear in τ

III $(\mathcal{K} \cup \{t_1 \mapsto [\![F_1]\!], t_2 \mapsto [\![F_2]\!]\}, E \cup \{(t_1, t_2)\}, S) \Rightarrow$
$([t_1 \mapsto t_2](\mathcal{K} \cup \{t_2 \mapsto [\![F]\!]\}),$
$[t_1 \mapsto t_2](E \cup \{(F_1(l), F_2(l)) | l \in domain(F_1) \cap domain(F_2)\}),$
$\{(t_1, t_2)\} \cup [t_1 \mapsto t_2](S))$
where $F = \{(l, \tau_l) | l \in domain(F_1) \cup domain(F_2),$
$\tau_l = F_1(l)$ if $l \in domain(F_1)$ otherwise $\tau_l = F_2(l)\}$
if t_1 not appears in F_2 and t_2 not appears in F_1.

IV $(\mathcal{K} \cup \{t_1 \mapsto [\![F_1]\!]\}, E \cup \{(t_1, [F_2])\}, S) \Rightarrow$
$([t_1 \mapsto [F_2]](\mathcal{K}),$
$[t_1 \mapsto [F_2]](E \cup \{(F_1(l), F_2(l)) | l \in domain(F_1) \cap domain(F_2)\}),$
$\{(t_1, [F_2])\} \cup [t_1 \mapsto [F_2]](S))$
if $domain(F_1) \subseteq domain(F_2)$ and $t \notin FTV([F_2])$

V $(\mathcal{K}, E \cup \{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2)\}, S) \Rightarrow (\mathcal{K}, E \cup \{(\tau_1^1, \tau_2^1), (\tau_1^2, \tau_2^2)\}, S)$

VI $(\mathcal{K}, E \cup \{[F_1], [F_2]\}, S) \Rightarrow (\mathcal{K}, E \cup \{(F_1(l), F_2(l)) | l \in domain(F_1)\}, S)$
if $domain(F_1) = domain(F_2)$

Fig. 7. Some of the Transformation Rules for Kinded Unification

$S(\mathcal{A}_1(t))$ for all $t \in domain(\mathcal{A}_1)$, and $\tau_2 = S(\tau_1)$. A typing $\mathcal{K}, \mathcal{A} \rhd e : \tau$ is *principal* if it is more general than all the derivable typings for e. The type inference algorithm is extended by adding the new functions to compose a principal type for record and variant operations and to extend the main algorithm by adding the cases for records and variants. Figure 8 shows the necessary changes to the main algorithm. Figure 9 shows the new composition functions corresponding to the typing rules for records and variants. The functions we have defined in Figure 3 remain unchanged except that they take kinded typings of the form $(\mathcal{K}, \mathcal{A}, \tau)$ and the appropriate kind assignments must be added as components of the parameter of the unification algorithm and of its result.

Figure 10 shows the type inference process for the function `fn x => (x.Name, x.Sal > 10000)`, a function that is used in the implementation of `Wealthy` of section 1.

3.5 Further Refinement and the Correctness of the Type Inference System

In the explanation of type inference algrithm so far, we have ignored the constraint that some type variables should only denote description types. The necessary extension is to introduce description kind constructors D, $[\![l : \delta, \ldots, l : \delta]\!]_d$ and $\langle\!\langle l : \delta, \ldots, l : \delta \rangle\!\rangle_d$ respectively denoting the set of all description types, description record types, and description variant types. Although it increases the notational complexity, these extension can be easily incorporated with the unification algorithm and the type inference.

Another simplification we made in the description of the type inference algorithm is our assumption that types are all non cyclic. To extend the type inference algorithm to recursive types, we only need to extend the kinded unification algorithm

$Typing(e, L) =$
  case e of
    c_τ $\Longrightarrow (\emptyset, \emptyset, \tau)$
    x $\Longrightarrow$ if $x \in domain(L)$ then $L(x)$ with all type variables renamed
      else $(\{t :: U\}, \{x : t\}, t)$ (t fresh)
    ⋮
    $[l_1{=}e_1,\ldots,l_n{=}e_n]$ $\Longrightarrow P_{\text{RECORD}}([l_1 = Typing(e_1, L), \ldots, l_n = Typing(e_n, L)])$
    $e.l$ $\Longrightarrow P_{\text{DOT}}(Typing(e, L), l)$
    modify(e_1,l,e_2) $\Longrightarrow P_{\text{MODIFY}}(Typing(e_1, L), Typing(e_2, L), l)$
    <l=e> $\Longrightarrow P_{\text{VARIANT}}(Typing(e, L), l)$
    case e of <l_1=x_1> => e_1,..., <l_n=x_n> => e_n endcase $\Longrightarrow$
      $P_{\text{CASE1}}(Typing(e, L),$
        $[l_1 = P_{\text{ABS}}(Typing(e_1, L), x_1), \ldots,$
        $l_n = P_{\text{ABS}}(Typing(e_n, L), x_n)])$
    case e of <l_1=x_1> => e_1,..., <l_n=x_n> => e_n else e_0 endcase $\Longrightarrow$
      $P_{\text{CASE2}}(Typing(e, L),$
        $[l_1 = P_{\text{ABS}}(Typing(e_1, L), x_1), \ldots,$
        $l_n = P_{\text{ABS}}(Typing(e_n, L), x_n)],$
        $Typing(e_0, L))$
  endcase

Fig. 8. The Main Algorithm for Type Inference with Records and Variants

to infinite regular trees. The necessary extension is similar to the one needed to extend an ordinary unification algorithm to regular trees [Huet 1976], which involves: (1) defining a data structure to represent regular trees; (2) changing the cases for variable elimination (cases of II and IV) by eliminating occur-check and replacing the one point substitution $[t \mapsto \tau]$ by the substitution $[t \mapsto \overline{(\text{rec } v.\tau[v/t])}]$ where $\overline{(\text{rec } v.\tau[v/t])}$ is a regular tree that is a solution to $v = \tau[v/t]$, and (3) changing the cases for decomposition (cases V and VI) so that they generate the equations for the set of pairs of corresponding subtrees of the given regular trees.

We have also ignored the details of dealing with references. The above type inference method cannot be directly extended to references, since the call-by-value operational semantics for let expressions involving references does not agree with polymorphic type discipline for let binding. As Milner observed in his original presentation of ML type system [Milner 1978], the straightforward application of ML type inference method to references yields unsound type system. Solutions have been proposed in [Tofte 1988; MacQueen 1988; Leroy and Weise 1991; Hoang et al. 1993]. They differ in detailed treatment but they are all based on the idea that the type system restricts substitution on type variables in reference types in such a way that references created by a polymorphic functions are monomorphic. Since these mechanisms can be regarded as a new form of kind constraint on type variables, we believe that they can safely be incorporated with our type system. Another more radical solution [Leroy 1993] is to change the semantics of let to call-by-name, with which Damas-Milner polymorphic let typing and equivalently our rule for let become sound. This strategy can also be adopted. However, for want of good intuitions about the merits of these mechanisms, we adopt the simplest and restrict the reference constructor to monomorphic types.

With these refinements, the complete static type inference of ML is extended to

$$
\begin{array}{l}
P_{\mathrm{RECORD}}([l_1 = (\mathcal{K}_1, \mathcal{A}_1, \tau_1), \ldots, l_n = (\mathcal{K}_n, \mathcal{A}_n, \tau_n)]) = \\
\quad \mathit{let}\ (\mathcal{K}, S) = \mathit{Unify}(\mathcal{K}_1 \cup \cdots \cup \mathcal{K}_n, \mathit{allpairs}(\{\mathcal{A}_1, \ldots, \mathcal{A}_n\})) \\
\quad \mathit{in}\ (\mathcal{K}, S(\mathcal{A}_1) \cup \cdots \cup S(\mathcal{A}_n), S([l_1 : \tau_1, \ldots, l_n : \tau_n])) \\
\quad \mathit{end}
\end{array}
$$

$$
\begin{array}{l}
P_{\mathrm{DOT}}((\mathcal{K}, \mathcal{A}, \tau), l) = \\
\quad \mathit{let}\ (\mathcal{K}', S) = \mathit{Unify}(\mathcal{K} \cup \{t_1 :: U, t_2 :: [\![l : t_1]\!]\}, \{(t_2, \tau)\}) \qquad (t_1, t_2 \text{ fresh}) \\
\quad \mathit{in}\ (\mathcal{K}', S(\mathcal{A}), S(t_1)) \\
\quad \mathit{end}
\end{array}
$$

$$
\begin{array}{l}
P_{\mathrm{MODIFY}}((\mathcal{K}_1, \mathcal{A}_1, \tau_1), (\mathcal{K}_2, \mathcal{A}_2, \tau_2), l) = \\
\quad \mathit{let}\ (\mathcal{K}, S) = \mathit{Unify}(\mathcal{K}_1 \cup \mathcal{K}_2 \cup \{t_1 :: U, t_2 :: [\![l : t_1]\!]\}, \\
\qquad\qquad \mathit{allpairs}(\{\mathcal{A}_1, \mathcal{A}_2\}) \cup \{(t_2, \tau_1), (t_1, \tau_2)\}) \qquad (t_1, t_2 \text{ fresh}) \\
\quad \mathit{in}\ (\mathcal{K}, S(\mathcal{A}), S(t_2)) \\
\quad \mathit{end}
\end{array}
$$

$$
P_{\mathrm{VARIANT}}((\mathcal{K}, \mathcal{A}, \tau), l) = (\mathcal{K} \cup \{t :: \langle\!\langle l : \tau \rangle\!\rangle\}, \mathcal{A}, t) \quad (t \text{ fresh})
$$

$$
\begin{array}{l}
P_{\mathrm{CASE1}}((\mathcal{K}_0, \mathcal{A}_0, \tau_0), [l_1 = (\mathcal{K}_1, \mathcal{A}_1, \tau_1), \ldots, l_n = (\mathcal{K}_n, \mathcal{A}_n, \tau_n)]) = \\
\quad \mathit{let}\ (\mathcal{K}, S) = \\
\qquad \mathit{Unify}(\mathcal{K}_0 \cup \cdots \cup \mathcal{K}_n \cup \{t :: U, t_1 :: U, \ldots, t_n :: U\}, \\
\qquad\qquad \mathit{allpairs}(\{\mathcal{A}_0, \ldots, \mathcal{A}_n\}) \cup \{(\tau_i, t_i \to t) | 1 \le i \le n\} \\
\qquad\qquad \cup \{(\tau_0, \langle l_1 : t_1, \ldots, l_n : t_n \rangle)\}) \quad (t, t_1, \ldots, t_n \text{ fresh}) \\
\quad \mathit{in}\ (\mathcal{K}, S(\mathcal{A}_1) \cup \cdots \cup S(\mathcal{A}_n), S(t)) \\
\quad \mathit{end}
\end{array}
$$

$$
\begin{array}{l}
P_{\mathrm{CASE2}}((\mathcal{K}_0, \mathcal{A}_0, \tau_0), [l_1 = (\mathcal{K}_1, \mathcal{A}_1, \tau_1), \ldots, l_n = (\mathcal{K}_n, \mathcal{A}_n, \tau_n)], (\mathcal{K}_{n+1}, \mathcal{A}_{n+1}, \tau_{n+1})) = \\
\quad \mathit{let}\ (\mathcal{K}, S) = \\
\qquad \mathit{Unify}(\mathcal{K}_0 \cup \cdots \cup \mathcal{K}_{n+1} \cup \{t :: U, t_1 :: U, \ldots, t_n :: U, t_0 :: \langle\!\langle l_1 : t_1, \ldots, l_n : t_n \rangle\!\rangle\}, \\
\qquad\qquad \mathit{allpairs}(\{\mathcal{A}_0, \ldots, \mathcal{A}_{n+1}\}) \cup \{(\tau_i, t_i \to t) | 1 \le i \le n\} \\
\qquad\qquad \cup \{(\tau_0, t_0), (\tau_{n+1}, t)\}) \qquad (t, t_0, t_1, \ldots, t_n \text{ fresh}) \\
\quad \mathit{in}\ (\mathcal{K}, S(\mathcal{A}_1) \cup \cdots \cup S(\mathcal{A}_n), S(t)) \\
\quad \mathit{end}
\end{array}
$$

Fig. 9. New Functions to Synthesize Principal Typings

$$
\begin{array}{l}
Typing(\mathsf{fn\ x\ =>\ (x.Name,\ x.Sal > 10000)}, \emptyset) \\
\quad = P_{\text{ABS}}(Typing((\mathsf{x.Name,\ x.Sal > 10000}), \emptyset), \mathsf{x}) \\
\quad\ \ \rangle Typing((\mathsf{x.Name,\ x.Sal > 10000}), \emptyset) \\
\quad\ \ \rangle \quad = P_{\text{RECORD}}((Typing(\mathsf{x.Name}, \emptyset), Typing(\mathsf{x.Sal > 10000}, \emptyset))) \\
\quad\ \ \rangle \quad\ \ \rangle Typing(\mathsf{x.Name}, \emptyset) \\
\quad\ \ \rangle \quad\ \ \rangle \quad = P_{\text{DOT}}(Typing(\mathsf{x}, \emptyset), \mathsf{Name}) \\
\quad\ \ \rangle \quad\ \ \rangle \quad\ \ \rangle Typing(\mathsf{x}, \emptyset) = (\{t_1 :: U\}, \{\mathsf{x} : t_1\}, t_1) \\
\quad\ \ \rangle \quad\ \ \rangle \quad = P_{\text{DOT}}((\{t_1 :: U\}, \{\mathsf{x} : t_1\}, t_1), \mathsf{Name}) \\
\quad\ \ \rangle \quad\ \ \rangle \quad = (\{t_2 :: U, t_1 :: [\![\mathsf{Name} : t_2]\!]\}, \{\mathsf{x} : t_1\}, t_2) \\
\quad\ \ \rangle \quad\ \ \rangle Typing(\mathsf{x.Sal > 10000}, \emptyset) \\
\quad\ \ \rangle \quad\ \ \rangle \quad = P_{>}(Typing(\mathsf{x.Sal}, \emptyset), Typing(10000, \emptyset)) \\
\quad\ \ \rangle \quad\ \ \rangle \quad\ \ \rangle Typing(\mathsf{x.Sal}, \emptyset) = (\{t_3 :: U, t_4 :: [\![\mathsf{Sal} : t_3]\!]\}, \{\mathsf{x} : t_4\}, t_3) \\
\quad\ \ \rangle \quad\ \ \rangle \quad\ \ \rangle Typing(10000, \emptyset) = (\emptyset, \emptyset, int) \\
\quad\ \ \rangle \quad\ \ \rangle \quad = P_{>}((\{t_3 :: U, t_4 :: [\![\mathsf{Sal} : t_3]\!]\}, \{\mathsf{x} : t_4\}, t_3), (\emptyset, \emptyset, int)) \\
\quad\ \ \rangle \quad\ \ \rangle \quad = (\{t_4 :: [\![\mathsf{Sal} : int]\!]\}, \{\mathsf{x} : t_4\}, bool) \\
\quad\ \ \rangle \quad = P_{\text{RECORD}}((\{t_2 :: U, t_1 :: [\![\mathsf{Name} : t_2]\!]\}, \{\mathsf{x} : t_1\}, t_2), \\
\qquad\qquad\qquad\qquad (\{t_4 :: [\![\mathsf{Sal} : int]\!]\}, \{\mathsf{x} : t_4\}, bool)) \\
\quad\ \ \rangle \quad = (\{t_2 :: U, t_1 :: [\![\mathsf{Name} : t_2, \mathsf{Sal} : int]\!]\}, \{\mathsf{x} : t_1\}, t_2 * bool) \\
\quad = P_{\text{ABS}}((\{t_2 :: U, t_1 :: [\![\mathsf{Name} : t_2, \mathsf{Sal} : int]\!]\}, \{\mathsf{x} : t_1\}, t_2 * bool), \mathsf{x}) \\
\quad = (\{t_2 :: U, t_1 :: [\![\mathsf{Name} : t_2, \mathsf{Sal} : int]\!]\}, \emptyset, t_1 \to t_2 * bool)
\end{array}
$$

Fig. 10. Examples of Type Inference with Records

records, variants, and set data types, as stated in the following result:

THEOREM 2. *Let e be any raw term of Machiavelli. If $Typing(e, \emptyset) = (\mathcal{K}, \mathcal{A}, \tau)$ then $\mathcal{K}, \mathcal{A} \rhd e : \tau$ is a principal typing of e. If $Typing(e, \emptyset)$ reports failure then e has no typing.*

Just as legal ML programs correspond to principal typing schemes with empty type assignment, legal Machiavelli programs correspond to principal kinded typing schemes with empty type assignment, ie. typings of the form $\mathcal{K}, \emptyset \rhd e : \tau$. Machiavelli prints a typing $\mathcal{K}, \emptyset \rhd e : \tau$ as

$$e : \tau'$$

where τ' is a type whose type variables are printed together with their kind constraints in $\mathcal{K}$ in the following formats:

—'a,'b,... for those type variables t such that $\mathcal{K}(t) = \mathsf{U}, \ldots$

—"a,"b,... for those description type variables d such that $\mathcal{K}(d) = \mathsf{D}, \ldots$

—'a::[l_1:τ_1,...,l_n:τ_n],... for those type variables t such that $\mathcal{K}(t) = [\![l_1 : \tau_1, \ldots, l_n : \tau_n]\!], \ldots$

—"a::[l_1:τ_1,...,l_n:τ_n],... for those description type variables d such that $\mathcal{K}(d) = [\![l_1 : \tau_1, \ldots, l_n : \tau_n]\!]_d, \ldots$

—'a::<l_1:τ_1,...,l_n:τ_n>,... for those type variables t such that $\mathcal{K}(t) = \langle\!\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle\!\rangle, \ldots$

—"a::<l_1:τ_1,...,l_n:τ_n>,... for those description type variables d such that $\mathcal{K}(d) = \langle\!\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle\!\rangle_d, \ldots$

as already seen in examples. Thus the type output in the following example

```
-> fun name x = x.Name;
>> val name = fn : 'a::[Name : 'b] -> 'b
```

is a representation of the following kinded typing scheme:

$$\{t_2 :: U, t_1 :: [\![\mathsf{Name} \ : \ t_2]\!]\}, \emptyset \rhd \mathsf{fn\ x=>\ x.Name} \ : \ t_1 \to t_2$$

The examples shown in Figure 1 use the same convention.

To summarize our progress to this point: we have augmented type schemes of ML with description types (which already exist in ML in a limited form) and kinded type variables. This has provided us with a type system that not only expresses the generic nature of field selection, but also allows sets to be uniformly treated in the language. However relational databases require more than the operations we have so far described, and it is to these that we now turn.

4. OPERATIONS FOR GENERALIZED RELATIONS

We are now going to show how we can extend Machiavelli to include the operations of the relational algebra, specifically *projection* and *natural join*, which are not covered by the operations developed so far. There are two points to be made about our strategy. The first is that we are going to put an ordering on values and on description types. The ordering on types, although somewhat similar to that used by Cardelli [Cardelli 1988], is in no sense a part of Machiavelli's polymorphism. This should be apparent from the fact that we have already incorporated polymorphic field selection without such an ordering.

The second point is that the introduction of *join* complicates the presentation of the type system and increases the complexity of the type inference problem, which requires us to extend the notion of (kinded) typing schemes to *conditional* typing schemes [Ohori and Buneman 1988] by adding syntactic conditions on instantiation of type variables. A similar problem was later observed in [Wand 1989] if one uses a record concatenation operation rather than join. (See also [Cardelli and Mitchell 1989; Harper and Pierce 1991; Remy 1992] for polymorphic calculi with record concatenation.) Since we are primarily concerned with database operations, our inclination is to examine the record joining operation that naturally arises as a result of generalizing the relational algebra.

Our strategy in this section is first to provide a method for generalizing relational algebra over arbitrary description types. We then provide the additional typing rules, which have associated order constraints on the types. We then provide a principal *conditional* typing scheme which represents the exact set of provable typings. Finally, we describe a method to check statically the satisfiability of these constraints. In other words, we are still able to guarantee that a typechecked program will not cause a run-time type error.

4.1 Generalizing Relational Algebra

Our rationale for wanting to generalize relational operations is that, in keeping with the rest of the language, we would like them to be as "polymorphic" as possible. Since equality is essential to the definition of most of these operations, we limit ourselves to their effect on description types. To this end we generalize the following four operations to arbitrary description terms and introduce them as polymorphic functions into the system:

$\mathsf{eq}(e_1,e_2)$	*equality test,*
$\mathsf{join}(e_1,e_2)$	*database join,*
$\mathsf{con}(e_1,e_2)$	*consistency check,*
$\mathsf{project}(e,\delta)$	*projection of d onto the type δ.*

The intuition underlying their generalization is the idea exploited in [Buneman et al. 1991] that database objects are *partial descriptions* of real-world entities and can be ordered by *goodness of description.* The polymorphic type system to represent generalized relational operations (including cyclic structures) has been developed in [Ohori 1990]. Here, we limit ourselves to acyclic description terms. We first consider join and equality.

We claim that join in the relational model is based on the underlying operation that computes a join of tuples. By regarding tuples as partial descriptions of real-world entities, we can characterize it as a special case of very general operations on partial descriptions that *combines* two consistent descriptions. For example, if we consider the following non-flat tuples

$$t_1 = [\mathsf{Name} = [\mathsf{First} = \text{"}\mathsf{Joe}\text{"}]] \quad \text{and} \quad t_2 = [\mathsf{Name} = [\mathsf{Last} = \text{"}\mathsf{Doe}\text{"}]]$$

as partial descriptions, then the combination of the two should be

$$t = [\mathsf{Name} = [\mathsf{First} = \text{"}\mathsf{Joe}\text{"}, \mathsf{Last} = \text{"}\mathsf{Doe}\text{"}]].$$

Thus t is the least upper bound $t_1 \sqcup t_2$ of t_1 and t_2 under an ordering $\sqsubseteq$ induced by the inclusion of record fields, so that

$$\mathsf{join}(d_1,d_2) = d_1 \sqcup d_2 \quad \text{and} \quad \mathsf{eq}(d,d') = d \sqsubseteq d' \text{ and } d' \sqsubseteq d$$

This approach also provides a uniform treatment of *null values* [Zaniolo 1984; Biskup 1981], which are used in databases that represent incomplete information. To represent null values, we also extend the syntax of Machiavelli terms with:

$\mathsf{null}(b)$	*the null value of a base type b,*
$\langle\rangle$	*the (polymorphic) null value of variant types.*

Other incomplete values can be built from these using the constructors for description terms. However care must be taken [Lipski 1979; Imielinski and Lipski 1984] to ensure that use of the algebra with these extended operations and null values provides the semantics intended by the programmer.

These characterizations do not depend on any particular data structure such as flat records. Once we have defined a (computable) ordering on the set of description terms, join and equality generalize to arbitrary description terms. To obtain such an ordering, we first define a pre-order $\preceq$ on acyclic description terms.

$$\begin{aligned}
c^b &\preceq c^b \quad \textit{for all constants } c^b \textit{ of type } b,\\
\mathsf{null}(b) &\preceq c^b \quad \textit{for all constants } c^b \textit{ of type } b,\\
\mathsf{null}(b) &\preceq \mathsf{null}(b) \quad \textit{for any base type } b\\
[l_1 = d_1,\ldots,l_n = d_n] &\preceq [l_1 = d'_1,\ldots,l_n = d'_n,\ldots] \quad \textit{if } d_i \preceq d'_i\ (1 \le i \le n),\\
\langle\rangle &\preceq \langle\rangle,\\
\langle\rangle &\preceq \langle l = d\rangle \quad \textit{for any description } d,\\
\langle l = d\rangle &\preceq \langle l = d'\rangle \quad \textit{if } d \preceq d',\\
r &\preceq r \quad \textit{for any reference } r,
\end{aligned}$$

```
r1 = {[Pname = "Nut",Supplier = { [Sname = "Smith",City = "London"],
                                  [Sname = "Jones",City = "Paris"],
                                  [Sname = "Blake",City = "Paris"]}],
      [Pname = "Bolt",Supplier = { [Pname = "Blake",City = "Paris"],
                                   [Sname = "Adams",City = "Athens"]}]}

r2 = {[Pname = "Nut",Supplier = {[City = "Paris"]},Qty = 100],
      [Pname = "Bolt",Supplier = {[City="Paris"]},Qty = 200]}

join(r1,r2) = {[Pname = "Nut",Supplier ={[Sname = "Jones",City = "Paris"],
                                        [Sname = "Blake",City = "Paris"]}, Qty = 100],
               [Pname = "Bolt",Supplier ={[Sname = "Blake",City = "Paris"]}, Qty = 200]}
```

Fig. 11. Natural join of higher-order relations

$$\{d_1,\ldots,d_n\} \preceq \{d'_1,\ldots,d'_m\} \quad \textit{if } \forall d' \in \{d'_1,\ldots,d'_m\}.\, \exists d \in \{d_1,\ldots,d_n\}.\, d \preceq d'$$

The last rule expresses how the ordering can be extended to sets. Because $\preceq$ fails to be anti-symmetric an ordering is obtained by taking induced equivalence relation and regarding a description term as a representative of its equivalence class. Thus we take $\sqsubseteq$ as the ordering induced by $\preceq$. Among representatives, there is a canonical one having the property that if it contains a set term then its members are pairwise incomparable, i.e. an anti-chain. Since $\sqsubseteq$ and $\sqcup$ are computable, our characterization of join and eq immediately gives their definitions on general description terms, which computes a canonical representative. con, which checks for the existence of a join is also computable. The equality (eq) is a generalization of structural equality to sets and null values. Figure 11 shows an example of a join of complex descriptions. The importance of this definition of join is that it is a faithful generalization of the join in the relational model. In [Buneman et al. 1991] it is shown that:

THEOREM 3. *If r_1, r_2 are first-normal form relations then* join(r_1,r_2) *is the natural join of r_1 and r_2 in the relational model.*

Projection in the relational model is defined as a projection on a set of labels. We generalize it to an operation which projects a complex description term onto some type that describes part of its structure, and we define projection as an operation specified by this type. Recall that the syntax of ground (variable free) description types is

$$\delta ::= \mathsf{unit} \mid b_d \mid [l{:}\delta,\ldots,l{:}\delta] \mid \langle l{:}\delta,\ldots,l{:}\delta\rangle \mid \{\delta\} \mid \mathsf{ref}(\tau) \mid (\mathsf{rec}\ v.\delta(v))$$

Projection is therefore an operation indexed by a ground description type. The operation project(x,δ) takes a value x whose type is "bigger" than δ and returns a value of type δ by a generalized form of projection onto that type. The following is a simple projection on a flat relation, which has the obvious result:

```
project({ [Name = "J. Doe", Age = 21, Salary = 21000],
          [Name = "S. Jones", Age = 31, Salary = 31000] },
        {[Name : string, Salary : int]})
```

By using the ordering we have just defined, projection can be specified as:

$$\mathsf{project}(x, \delta) = \bigsqcup\{d | d \sqsubseteq x, d : \delta\}$$

which is computable for any description type δ.

These definitions for join and projection can be generalized to cyclic structures, and there are polymorphic algorithm for the generalized definitions [Ohori 1990].

To summarize these extensions to the language, we have introduced the constants $\mathsf{null}(b)$ and <> and the term constructors join, con, and project. Machiavelli's call-by-value operational semantics described in section 3 is directly extended to those operation by adding their computation algorithms to the evaluation rules.

4.2 Type Inference for Relational Algebra

join, project and con are polymorphic operations in the sense that they compute join and projection of various types. To represent this, we define an ordering on ground description types that represents the ordering on the structure of descriptions. For the set of acyclic description types, this is given by the following inductive definition:

$$\begin{aligned}
b_d &\ll b_d \\
[l_1{:}\delta_1,\ldots,l_n{:}\delta_n] &\ll [l_1{:}\delta'_1,\ldots,l_n{:}\delta'_n,\ldots] \text{ if } \delta_i \ll \delta'_i \ (1 \le i \le n) \\
\langle l_1{:}\delta_1,\ldots,l_n{:}\delta_n\rangle &\ll \langle l_1{:}\delta'_1,\ldots,l_n{:}\delta'_n\rangle \text{ if } \delta_i \ll \delta'_i \ (1 \le i \le n) \\
\{\delta_1\} &\ll \{\delta_2\} \text{ if } \delta_1 \ll \delta_2 \\
\mathsf{ref}(\delta) &\ll \mathsf{ref}(\delta)
\end{aligned}$$

This definition reflects the definition of the ordering on description terms. In particular, the rule for reference types reflects the property that two reference expressions have a join if they denote the same reference value, and therefore have the same type. (There may be other choices for the ordering of variant types. The choice will depend on the intended semantics of this construct as a partial description.) Using this ordering, types of join, project, and con are given as:

$$\begin{aligned}
\mathsf{con} &: \delta_1 * \delta_2 \to \mathit{bool} \text{ such that } \delta_1 \sqcup_{\ll} \delta_2 \text{ exsists} \\
\mathsf{join} &: \delta_1 * \delta_2 \to \delta_3 \text{ such that } \delta_3 = \delta_1 \sqcup_{\ll} \delta_2 \\
\mathsf{project}(_, \delta_2) &: \delta_1 \to \delta_2 \text{ such that } \delta_2 \ll \delta_1
\end{aligned}$$

To integrate these operations with the polymorphic core of Machiavelli defined in section 3, we need to take account of these operations in the type system. We therefore explicitly introduce syntactic conditions on substitution of type variables for three forms of constraint associated with the types of these operations: $\delta_1 \sqcup_{\ll} \delta_2$ exists, $\delta = \delta_1 \sqcup_{\ll} \delta_2$, and $\delta_2 \ll \delta_1$. In fact we only need to consider the last two forms of constraint since $\delta_1 \sqcup_{\ll} \delta_2$ will exist whenever we can find a type $\delta_3 = \delta_1 \sqcup_{\ll} \delta_2$. To represent them we introduce the following syntactic conditions:

(1) $\tau = \mathit{jointype}(\tau, \tau)$, and
(2) $\mathit{lessthan}(\tau, \tau)$.

Note the difference between $\delta_3 = \delta_1 \sqcup_{\ll} \delta_2$ and $\tau_3 = \mathit{jointype}(\tau_1, \tau_2)$. The former is a property on the relationship between three ground description types. On the other hand, the latter is a syntactic formula denoting the constraint on substitutions

$$(\text{NULL1}) \qquad C, \mathcal{K}, \mathcal{A} \rhd \mathsf{null}(b) \ : \ b$$

$$(\text{NULL2}) \qquad C, \mathcal{K}, \mathcal{A} \rhd \langle\rangle \ : \ \delta \qquad \text{if } \mathcal{K} \vdash \delta :: \langle\langle\rangle\rangle$$

$$(\text{CON}) \qquad \frac{C, \mathcal{K}, \mathcal{A} \rhd e_1 \ : \ \delta_1 \qquad C, \mathcal{K}, \mathcal{A} \rhd e_2 \ : \ \delta_2}{C \cup \{\delta = \mathit{jointype}(\delta_1, \delta_2)\}, \mathcal{K}, \mathcal{A} \rhd \mathsf{con}(e_1, e_2) \ : \ \mathit{bool}} \qquad (\text{for some } \delta)$$

$$(\text{JOIN}) \qquad \frac{C, \mathcal{K}, \mathcal{A} \rhd e_1 \ : \ \delta_1 \qquad C, \mathcal{K}, \mathcal{A} \rhd e_2 \ : \ \delta_2}{C \cup \{\delta = \mathit{jointype}(\delta_1, \delta_2)\}, \mathcal{K}, \mathcal{A} \rhd \mathsf{join}(e_1, e_2) \ : \ \delta}$$

$$(\text{PROJECT}) \qquad \frac{C, \mathcal{K}, \mathcal{A} \rhd e \ : \ \delta_1}{C \cup \{\mathit{lessthan}(\delta_2, \delta_1)\}, \mathcal{K}, \mathcal{A} \rhd \mathsf{project}(e_1, \delta_2) \ : \ \delta_2}$$

Fig. 12. The Typing Rules for Relational Operations

of type variables in τ_1, τ_2, τ_3 to ensure that any ground instance of these satisfies such a property. A similar remark holds for $\delta_1 \ll \delta_2$ and $\mathit{lessthan}(\tau_1, \tau_2)$. Using these syntactic conditions on type variables, we can extend the type system to incorporate these new operations. Typing judgements in the extended system has the form $C, \mathcal{K}, \mathcal{A} \rhd e \ : \ \tau$ where the extra ingredient C is a set of these syntactic conditions. Figure 12 shows the typing rules for the new operations. Other rules remain the same as those defined in Figure 2 and 6 except that they are now relative to a given set of conditions. For example, the rule ABS becomes

$$(\text{ABS}) \qquad \frac{C, \mathcal{K}, \mathcal{A}(x, \tau_1) \rhd e \ : \ \tau_2}{C, \mathcal{K}, \mathcal{A} \rhd \mathsf{fn}\ x\ \mathsf{=>}\ e \ : \ \tau_1 \rightarrow \tau_2}$$

In particular, these other rules only propagate the given set of conditions and do not change its contents.

Since the conditions we introduced involve the ordering that is defined only on ground types, we need to interpret a typing judgement in this extended system as a *scheme* representing the set of all ground typings obtained by substituting its type variables with appropriate ground types. This interpretation is consistent with our treatment of let construct (LET rule in Figure 2) and its semantics described in [Ohori 1989a]. A ground substitution θ *satisfies* a condition c if

(1) if $c \equiv \tau_1 = \mathit{jointype}(\tau_2, \tau_2)$ then $\theta(\tau_1), \theta(\tau_2), \theta(\tau_3)$ are description types satisfying $\theta(\tau_1) = \theta(\tau_2) \sqcup_{\ll} \theta(\tau_3)$,
(2) if $c \equiv \mathit{lessthan}(\tau_1, \tau_2)$ then $\theta(\tau_1)$ and $\theta(\tau_2)$ are description types satisfying $\theta(\tau_1) \ll \theta(\tau_2)$.

θ satisfies a set C of conditions if it satisfies each member of C. We say that a ground typing $\emptyset, \emptyset, \mathcal{A} \rhd e \ : \ \tau$ is an instance of $C, \mathcal{K}, \mathcal{A}' \rhd e \ : \ \tau'$ if there is a ground substitution θ that respects $\mathcal{K}$ and satisfies C such that $\mathcal{A} \uparrow^{dom(\mathcal{A}')} = \theta(\mathcal{A}')$ and $\tau = \theta(\tau')$. As seen in this definition, a typing in the extended system is subject to a set of conditions associated with it. To emphasize this fact, we call typing judgement in the extended type system a *conditional typing*. A conditional typing scheme $C, \mathcal{A} \rhd e \ : \ \tau$ is *principal* if any derivable ground typing for e is an instance of it. The following result establishes the complete inference of principal conditional typing schemes.

THEOREM 4. *There is an algorithm which, given any raw term e, returns either* failure *or a tuple* $(C, \mathcal{K}, \mathcal{A}, \tau)$ *such that if it returns* $(C, \mathcal{K}, \mathcal{A}, \tau)$ *then* $C, \mathcal{K}, \mathcal{A} \rhd e \ : \ \tau$

```
-> fun join3(x,y,z) = join(x,join(y,z));
>> val join3 = fn : ("a * "b * "c) -> "d
   where { "d = jointype("a,"e), "e = jointype("b,"c) }
-> Join3([Name = "Joe"],[Age = 21],[Office = 27]);
>> val it = [Name = "Joe",Age = 21,Office = 27] : [Name : string,Age : int,Office : int]
-> project(it,[Name : string]);
>> val it = [Name="Joe"] : [Name : string]
```

Fig. 13. Some Simple Relational Examples

is a principal conditional typing scheme, otherwise e has no typing.

A proof of this, which also gives the type inference algorithm for Machiavelli, is based on the technique we have developed in [Ohori and Buneman 1988] which established the theorem for a sublanguage of Machiavelli. A complete proof and a complete type inference algorithm can be reconstructed from the corresponding proof and algorithm presented in [Ohori 1989b].

Figure 13 gives two simple examples of the typing schemes that are inferred by Machiavelli. The type ("a * "b * "c) -> "d where { "d = jointype("a,"e), "e = jointype("b,"c) } of the three-way join join3 is the representation of the principal conditional typing scheme:

$$C, K, \emptyset \rhd \mathsf{fn(x,y,z) => join(x,join(y,z))} \ : \ (d_2 * d_4 * d_5) \rightarrow d_1$$

where C and K are as follows.

$$C = \left\{ \begin{array}{l} d_1 = jointype(d_2, d_3), \\ d_3 = jointype(d_4, d_5) \end{array} \right\}, \quad K = \left\{ \begin{array}{l} d_1 :: D, \\ d_2 :: D, \\ d_3 :: D, \\ d_4 :: D, \\ d_5 :: D \end{array} \right\}$$

It is therefore tempting to identify legal Machiavelli programs with principal conditional typing schemes. There is however one problem in this approach: a conditional typing schemes may not have a satisfiable instance. In such a case, the term has no typing and should therefore be regarded as a term with type error. Unfortunately checking the satisfiability of a set of these conditions is NP-hard [Ohori and Buneman 1988]. A practical solution is to *delay* the satisfiability check of a set of conditions until its type variables are fully instantiated. Once the types of all type variables in a condition are known, its satisfiability can be efficiently checked and it can then be eliminated. To achieve complete static typechecking under this strategy, the type system must satisfy the following property: for any expression containing joins and projections, if its evaluation involves evaluation of joins and projections, then the conditions associated with the types of the joins and the projections in the expression only contain ground types. In most cases where type variables originate from polymorphic function definitions, this condition is satisfied. Since joins and projections appearing in the body of a polymorphic function are evaluated only after the parameters to these operations are bound to some values, type variables originating from the polymorphic function definition should

```
-> parts;
>> val it = {[Pname="bolt",P#=1,Pinfo=<Base= [Cost=5]>],
                ...
            [Pname="engine",P#=2189,
             Pinfo= <Composite = [SubParts={[P#=1,Qty=189], ...},
                                  AssemCost=1000]>],...}
        : {[Pname : string,P# : int,
            Pinfo : <Base : [Cost : int],
                     Composite : [SubParts : {[P# : int,Qty : int]},AssemCost : int]>]}
-> suppliers;
>> val it ={[Sname="Baker",S#=1,City="Paris"],...}
        : {[Sname : string,S# : int,City : string]}
-> supplied_by;
>> val it = {[P#=1,Suppliers={[S#=1],[S#=12],...}],...}
        : {[P# : int,Suppliers : {[S# : int]}]}
```

Fig. 14. A Part-Supplier Database in Generalized Relational Model

have all been instantiated with ground types in any expressions that cause evaluation of those joins and projections. However, there is one exception, which is the case of expressions containing joins and projections of variants. Since variants are themselves polymorphic values, their type variables may never be instantiated. To preserve complete static typechecking, we place the restriction that the programmer must supply the type specification of variants if they are arguments of join and projection (directly or indirectly through function abstraction and function application.) The type system can easily enforce this restriction by rejecting expressions whose conditions involves type variables having a variant kind. We therefore identify legal Machiavelli programs with principal conditional typing schemes where the only conditions are those that contain type variables that have a kind other than variant kinds. This approach yields a practical solution to typechecking polymorphic programs involving join and projection. Since the additional restriction we imposed does not restrict polymorphic function definitions, we believe that this method preserves most of the advantages of polymorphic typing without incurring algorithmic difficulty in checking satisfiability of conditions.

Figure 14 shows an example of a database containing non-flat records, variants, and nested sets, where we assume that the names parts, suppliers, and supplied_by are already bound. With the availability of a generalized join and projection, we can immediately write programs that manipulate such databases. Figure 15 shows some simple query processing for the database example in figure 14. Note the use of join and other relational operations on "non-flat" relations. Data and operations can be freely mixed with other features of the language including recursion, higher-order functions, and polymorphism. This allows us to write, with relative ease, powerful programs whose type correctness is checked at compile time. Figure 16 shows query processing on the example database using polymorphic functions. The function cost taking a part record and a set of such records as arguments computes the total cost of the part. In the case of a composite part, it first generates a set of records, each consisting of a subpart number and its cost, and then uses hom

```
(*Select all base parts *)
-> join(parts,{[Pinfo=<Base=[]>]});
>> val it = {[Pname="bolt", P#=1, Pinfo=<Base=[Cost=5]>],...}
       : {[Pname : string,P# : int,
           Pinfo : <Base : [Cost : int],
                   Composite : [SubParts : {[P# : int,Qty : int]}, AssemCost : int]>]}

(*List part names supplied by "Baker" *)
-> select x.Pname
   from x <- join(parts,supplied_by)
   where Join3(x.Suppliers,suppliers,{[Sname="Baker"]}) <> {};
>> {"bolt",...} : {string}
```

Fig. 15. Some Simple Queries

to accumulate the costs of subparts. In order to prevent the set constructor from collapsing subpart costs which are equal, the computed subpart cost is paired with the subpart number. Note that scope of type variables is limited to a single type scheme, so that instantiations of "a in the type of cost are independent of those of "a in the type of expensive-parts. Also, the apparent complexity of the type of cost could be reduced by naming the large repeated sub-expression.

Without proper integration of the data model and programming language, defining such a function and checking its type consistency is problematic. Moreover, the functions cost and expensive_parts are both parameterized by the relation (partdb) and their polymorphism allows them to be applied to many different types. This is particularly useful when we have several different databases with the same structure of cost information. Even if these databases differ in the structure of other information, these functions are uniformly applicable.

5. HETEROGENEOUS SETS

The previous section provided an extension to a polymorphic type system for records that enabled us to infer the type-correctness of programs that involve operations of the relational algebra – notably *projection* and *natural join*. Here, we shall use closely related mechanisms to deal with a problem that arises in object-oriented databases, that of dealing with *heterogeneous collections*. The problem arises from two apparently contradictory uses of inheritance that arise in programming languages and in databases. In object-oriented languages the term describes code sharing: by an assertion that *Employee* inherits from *Person* we mean that the methods defined for the class *Person* are also applicable to instances of the class *Employee*. In databases – notably in data modeling techniques – we associate sets $Ext(Person)$ and $Ext(Employee)$ with the entities *Person* and *Employee* and the inheritance of *Employee* from *Person* specifies set inclusion: $Ext(Employee) \subseteq Ext(Person)$.

These notions of inheritance are apparently contradictory. For example, if members of $Ext(Employee)$ are instances of *Employee*, how can they be members of $Ext(Person)$ whose members must all be instances of *Person*? One way out of this is to relax what we mean by "instance of" and to allow an instance of *Employee* also to be an instance of *Person*. We can now take $Ext(Person)$ as a heteroge-

```
(*a function to compute the total cost of a part *)
-> fun cost(p,partdb) =
        case p.Pinfo of
             <Base = x> => x.Cost,
             <Composite = x> =>
               hom(fn(y)=> y.SubpartsCost,+,0,
                    select [SubpartsCost=cost(z,partdb) * w.Qty,P#=w.P#]
                    from w <- x.SubParts, z <- partdb
                    where eq(z.P#,w.P#)) + x.AssemCost
        endcase;

>> val cost = fn
        : ("a::[Pinfo : <Base : "b::[Cost : int],
                        Composite : "c::[SubParts : {"d::[P# : "e,Qty : int]},
                                         AssemCost : int]>,
             P# : "e]
          * {"a::[Pinfo : <Base : "b::[Cost : int],
                          Composite : "c::[SubParts : {"d::[P# : "e,Qty : int]},
                                           AssemCost : int]>,
               P# : "e]})
          -> int

(*select names of "expensive" parts *)
-> fun expensive_parts(partdb,n) = select x.Pname
                                   from x <- partdb
                                   where cost(x,partdb) > n;
>> val expensive_parts = fn :
        : ({"a::[Pinfo : <Base : "b::[Cost : int],
                         Composite : "c::[SubParts : {"d::[P# : "e,Qty : int]},
                                          AssemCost : int]>,
             P# : "e, Pname : "f]}
          * int) -> {"f}

-> expensive_parts(parts,1000);
>> val it = {"engine",...} : {string}
```

Fig. 16. Query Processing Using Polymorphic Functions

neous set, some of whose members are also instances of *Employee*. Type systems, however, can make the manipulation of heterogeneous collections difficult or impossible by "losing" information. For example if l has type $list(Person)$ and e has type *Employee*, the result of $insert(e, l)$ will still have type $list(Person)$, and the first element of this list will only have type *Person*. This problem appears both in languages with a subsumption rule [Cardelli 1988] and in statically type-checked object-oriented languages such as C++ [Stroustrup 1987] which claim the ability to represent heterogeneous collections as an important feature. In some cases the information is not recoverable; in others it can only be recovered in a rather dangerous fashion by "type-casting" values on the basis of information maintained by the programmer. A preliminary solution to this problem was described by the authors in [Buneman and Ohori 1991]. The approach here is simplified by use of the techniques developed in the preceding sections.

5.1 Dynamic and partial values

We shall exploit an idea of *dynamic* values proposed in [Cardelli 1986]. These are values that carry their type with them, and can be regarded as a pair consisting of a type and a value of that type. A formal system for type systems with *dynamic* was developed in [Abadi et al. 1991]. In these proposals there are two operations on dynamic values; at any type τ we have:

dynamic : τ -> dynamic
coerce(τ) : dynamic -> τ

The function dynamic creates a value of type dynamic out of a value of any type – operationally it pairs the value with its type. Conversely coerce(τ) takes such a pair and returns the value component provided the type component is τ; otherwise it raises an exception. A standard use for dynamic values is for representing persistent data, since the type of external data cannot be guaranteed. For example 2 + coerce(int)(read(input_stream)) will either add 2 to the input or raise an exception.

Our approach to heterogeneous collections is to generalize the notion of a dynamic type to one in which *some* of the structure is visible. A type $\mathcal{P}$([Name : string, Age : int]) denotes dynamic values whose actual type δ is "bigger" than [Name : string, Age : int], i.e. [Name : string, Age : int] $\ll \delta$ where $\ll$ is the ordering we used to represent types of relational operators. Thus the assertion $e : \mathcal{P}$([Name : string, Age : int]) means that e is a dynamic value, but it is known to be a record and that at least Name and Age fields are available on e. We shall refer to such partially specified dynamic values as *partial values*. Note that a partial value is like a dynamic value in that it always carries its (complete) type. The new type constructor $\mathcal{P}$ allows us to mix those partial values with other term constructors in the language. For example, $e' : \{\mathcal{P}(\delta)\}$ means that e' is a set of objects each of which is a partial value whose complete type is bigger than δ (under the ordering $\ll$.) It is this use of the ordering on types in conjunction with a set type that allows us to express heterogeneous collections. An assertion of the form $e : \{\mathcal{P}$([Name : string, Age : int])$\}$ means that e is a set of records, each of which has at least a Name : string and Age : int field, and therefore relational queries involving only selection of these fields are legitimate. As a special case of partial types, we introduce a constant type any denoting dynamic values on which no information is known – it is a (completely) dynamic value.

To show the use of partial types, let us assume that the following names have been given for partial types:

Person$*$ for $\mathcal{P}$([Name : string, Address : string])
Employee$*$ for $\mathcal{P}$([Name : string, Address : string, Salary : int])
Customer$*$ for $\mathcal{P}$([Name : string, Address : string, Balance : int])

Also suppose that DB is a set of type {any} so that we initially have no information about the structure of members of this set. Here are some examples of how such a database may be manipulated in a type-safe language.

(1) An operation filter $\mathcal{P}(\delta)$ (S) can be defined, which selects all the elements of S which have partial type $\mathcal{P}(\delta)$, i.e. filter $\mathcal{P}(\delta)$ $(S) : \{\mathcal{P}(\delta)\}$. We may use this in a query such as

```
select [Name=x.Name, Address=x.Address]
from x <- filter Employee* (DB)
where x.Salary > 10,000
```

The result of this query is a set of (complete) records, i.e. a relation. There is some similarity with the $*$ form of Postgres [Stonebraker and Rowe 1986], however we may use filter on arbitrary kinds and heterogeneous sets; we are not confined to the extensionally defined relations in the database.

(2) Under our interpretation of partial types, if $\delta_1 \ll \delta_2$ then $\mathcal{P}(\delta_1)$ is more partial than $\mathcal{P}(\delta_2)$ and any partial value of type $\mathcal{P}(\delta_2)$ also has type $\mathcal{P}(\delta_1)$. This property can be used to represent the desired set inclusion in the type system. In particular, Person$*$ is more partial than Employee$*$. From this, the inclusion filter Employee$*$ $(S) \subseteq$ filter Person$*$ (S) will always hold for any heterogeneous set S, in particular for the database DB. Thus the "data model" (inclusion) inheritance is *derived* from a property of type system rather than being something that must be achieved by the explicit association of extents with classes.

(3) By modifying the technique used to give a polymorphic type of join, we can define the typing rules for unions and intersections of heterogeneous sets. By adding a partial type any, the partialness ordering has meet and join operations. The union and intersection of heterogeneous sets have, respectively, the meet and join of their partial types. Thus, the type system can infer an appropriate partial type of heterogeneous set obtained by various set operations. For example, the following typings are inferred.

```
union(filter Customer* (DB), filter Employee* (DB))
  : {P([Name : string, Address : string])}

intersection(filter Customer* (DB), filter Employee* (DB))
   : {P([Name : string, Address : string, Salary : int, Balance : int])}.
```

(intersection is definable in the language) These inferred types automatically allow appropriate polymorphic functions to be applied to the result of these set operations. For example, since the type of an intersection of two heterogeneous sets is the join of the types, polymorphic functions applicable to either of the two sets are applicable to the intersection. Thus, we successfully achieve the desired coupling of set inclusion and method inheritance.

(4) We have the ability to write functions such as

```
fun RichCustomers(S) = select [Name=x.Name, Balance=x.Balance]
                       from x <- intersection(S,filter Customer* (DB))
                       where x.Salary > 30,000
```

Type inference allows the application of this function to any heterogeneous set each members of which has at least the type $\mathcal{P}$([Salary : int]). The result is a uniformly typed set, i.e. a set of type {[Name : string, Balance : int]}. Thus the application RichCustomers(filter Employee* (DB)) is valid, but the application RichCustomers(filter Customer* (DB)) does not have a type, and this will be statically determined by the failure of type inference.

In the following subsections we shall describe the basic operations for dealing with sets and partial values. We shall then give typing rules to extend Machiavelli to include those partial values.

5.2 The Basic Operations

To deal with partial values we introduce four new primitive operations: dynamic, as, coerce and fuse. We also extend the meaning of some of the existing primitives, such as union.

dynamic(e). This is used to construct a partial value and has type $\mathcal{P}(\delta)$ where δ is the type of e. A heterogeneous set may be constructed with

```
{dynamic([Name = "Joe", Age = 10]), dynamic([Name = "Jane", Balance = 10954])}
```

This expression implicitly makes use of union, and as a result of the extended typing rules for union, the expression has type $\{\mathcal{P}([\text{Name} : \text{string}])\}$, which is the meet of $\{\mathcal{P}([\text{Name} : \text{string}, \text{Age} : \text{int}])\}$ and $\{\mathcal{P}([\text{Name} : \text{string}, \text{Balance} : \text{int}])\}$.

The remaining three primitives may all fail. Rather than introduce an exception handling mechanism, we adopt the strategy that if the operation succeeds, we return the result in a singleton set, and if it fails, we return the empty set.

as $\mathcal{P}(\delta)$ (e). This, for any description type δ, "exposes" the properties of e specified by the type δ. This returns a singleton set containing the partial value if the coercion is possible and the empty set if it is not. For example, if e = as $\mathcal{P}$([Name : string]) (dynamic([Name = "Joe", Balance = 43.21])), e will have partial type $\{\mathcal{P}([\text{Name} : \text{string}])\}$ and an expression such as select x.Name from x <- e will type check, while select x.Balance from x <- e will not.

Using as and hom we are now in a position to construct the filter operation, mentioned earlier, which ties the inclusion of extents to the ordering on types. Because we do not have type parameters, it cannot be defined in the language. However it can be treated as a syntactic abbreviation:

$$\text{filter } \mathcal{P}(\delta)\ (S) \equiv \text{hom}(\text{fn x => as } \mathcal{P}(\delta)\ (\text{x}),\text{union}, S, \{\})$$

coerce δ (e). This coerces the partial value denoted by e to a (complete) value of type δ. It will only succeed if the type component of e is δ. Again, if the operation succeeds we return the singleton set, otherwise we return the empty set. For example coerce [Name : string] (dynamic([Name = "Jane", Balance = 10954])) will yield the empty set while coerce [Name : string, Balance : int] (dynamic([Name = "Jane", Balance = 10954])) will return the set {[Name = "Jane", Balance = 10954]}

fuse(e_1, e_2). This combines the partial values denoted by e_1 and e_2. It will only succeed if the (complete) values of e_1 and e_2 are equal. If e_1 has partial type $\mathcal{P}(\delta_1)$ and e_2 has partial type $\mathcal{P}(\delta_2)$ then fuse(e_1, e_2) will have the partial type $\mathcal{P}(\delta_1 \sqcup_{\ll} \delta_2)$. If

e_1 =(dynamic([Name = "Jane", Age = 21, Balance = 10954])),
e_2 = as $\mathcal{P}$([Name : string]) e_1,
e_3 = as $\mathcal{P}$([Age : int]) e_1, and
e_4 = as $\mathcal{P}$([Name : string]) dynamic([Name = "Jane"]),

then fuse(e_2, e_3) will be a singleton set of type $\{\mathcal{P}$([Name : string, Age : int])$\}$ while fuse(e_2, e_4) will return the empty set. fuse may be used to define set intersection as in

```
fun fuse1(x,s) = hom(fn y => fuse(x,y), union, s, {})
fun intersection(s1,s2) = hom(fn y => fuse1(y,s2), union, s1, {})
```

Note that fuse is more basic than equality for we can compute whether the partial values v_1 and v_2 are equal (as complete values) by not(empty(fuse(v_1, v_2))).

5.3 Extension of the Language

To incorporate these partial values, we extend the definition of the language. The set of types is extended to include any and the partial type constructor $\mathcal{P}(\delta)$:

$\tau ::= \cdots \mid$ any $\mid \mathcal{P}(\delta)$

We identify the following subset (ranged over by π) that may contain partial types.

$\pi ::=$ d $\mid b_d \mid [l{:}\pi,\ldots,l{:}\pi] \mid \langle l{:}\pi,\ldots,l{:}\pi\rangle \mid \{\pi\} \mid$ ref$(\pi) \mid$ any $\mid \mathcal{P}(\delta)$

The set of terms is extended to include operations for partial values.

$e ::= \cdots \mid$ dynamic$(e) \mid$ fuse$(e,e) \mid$ as $\mathcal{P}(\delta)$ $e \mid$ coerce δ e

To extend the type system to these new term constructors for partial values, we define an ordering on the above subset of types, which represents the partialness of types. We write $\pi \lesssim \pi'$ to denote that π is more partial than π'. The rules to define this ordering are:

$$\begin{aligned}
\text{any} &\lesssim \mathcal{P}(\delta) \text{ for any } \delta \\
\mathcal{P}(\delta_1) &\lesssim \mathcal{P}(\delta_2) \quad \text{if } \delta_1 \ll \delta_2 \\
b_d &\lesssim b_d \\
[l_1{:}\pi_1,\ldots,l_n{:}\pi_n] &\lesssim [l_1{:}\pi'_1,\ldots,l_n{:}\pi'_n] \quad \text{if } \pi_i \lesssim \pi'_i \ (1 \le i \le n) \\
\langle l_1{:}\pi_1,\ldots,l_n{:}\pi_n\rangle &\lesssim \langle l_1{:}\pi'_1,\ldots,l_n{:}\pi'_n\rangle \quad \text{if } \pi_i \lesssim \pi'_i \ (1 \le i \le n) \\
\{\pi\} &\lesssim \{\pi'\} \quad \text{if } \pi \lesssim \pi' \\
\text{ref}(\pi) &\lesssim \text{ref}(\pi') \quad \text{if } \pi \lesssim \pi'
\end{aligned}$$

The first two of these rules derive the order on partial types directly from the ordering $\ll$ that we introduced in section 4. The remaining rules lift this ordering component-wise to all description types. The following are examples of this ordering.

$$(\text{DYNAMIC}) \quad \frac{C, \mathcal{K}, \mathcal{A} \rhd e \ : \ \delta}{C, \mathcal{K}, \mathcal{A} \rhd \mathsf{dynamic}(e) \ : \ \mathcal{P}(\delta)}$$

$$(\text{AS}) \quad \frac{C, \mathcal{K}, \mathcal{A} \rhd e \ : \ \mathcal{P}(\delta)}{C, \mathcal{K}, \mathcal{A} \rhd \ \mathsf{as}\ \mathcal{P}(\delta')\ e \ : \ \{\mathcal{P}(\delta')\}}$$

$$(\text{COERCE}) \quad \frac{C, \mathcal{K}, \mathcal{A} \rhd e \ : \ \mathcal{P}(\delta)}{C, \mathcal{K}, \mathcal{A} \rhd \mathsf{coerce}\ \delta'\ e \ : \ \{\delta'\}}$$

$$(\text{FUSE}) \quad \frac{C, \mathcal{K}, \mathcal{A} \rhd e_1 \ : \ \pi_1 \qquad C, \mathcal{K}, \mathcal{A} \rhd e_2 \ : \ \pi_2}{C \cup \{\pi = \mathit{jointype}_{\lesssim}(\pi_1, \pi_2)\}, \mathcal{K}, \mathcal{A} \rhd \mathsf{fuse}(e_1, e_2) \ : \ \{\pi\}}$$

$$(\text{UNION}) \quad \frac{C, \mathcal{K}, \mathcal{A} \rhd e_1 \ : \ \{\pi_1\} \qquad C, \mathcal{K}, \mathcal{A} \rhd e_2 \ : \ \{\pi_2\}}{C \cup \{\pi = \mathit{meettype}_{\lesssim}(\pi_1, \pi_2)\}, \mathcal{K}, \mathcal{A} \rhd \mathsf{union}(e_1, e_2) \ : \ \{\pi\}}$$

Fig. 17. Typing Rules for Partial Values

$\mathcal{P}([\mathsf{Name} : \mathsf{string}, \mathsf{Address} : \mathsf{string}]) \lesssim \mathcal{P}([\mathsf{Name} : \mathsf{string}, \mathsf{Address} : \mathsf{string}, \mathsf{Balance} : \mathsf{int}])$

$[\mathsf{Acc\_No} : \mathsf{int}, \mathsf{Customer} : \mathcal{P}([\mathsf{Name} : \mathsf{string}, \mathsf{Address} : \mathsf{string}, \mathsf{Balance} : \mathsf{int}])] \lesssim$
$[\mathsf{Acc\_No} : \mathsf{int}, \mathsf{Customer} : \mathcal{P}([\mathsf{Name} : \mathsf{string}, \mathsf{Address} : \mathsf{string}, \mathsf{Balance} : \mathsf{int}, \mathsf{Salary} : \mathsf{int}])]$

Figure 17 gives the typing rules for the new term constructors. The new condition $\pi = \mathit{jointype}_{\lesssim}(\pi_1, \pi_2)$ used in rules (FUSE) denotes the condition on the ground substitutions θ such that $\theta(\pi) = \theta(\pi_1) \sqcup_{\lesssim} \theta(\pi_2)$, and the condition $\pi = \mathit{meettype}_{\lesssim}(\pi_1, \pi_2)$ used in the rule (UNION) denotes the ground substitutions θ such that $\theta(\pi) = \theta(\pi_1) \sqcap_{\lesssim} \theta(\pi_2)$.

Standard elimination operations introduced in Section 2 and database operations we defined in Section 4 are not available on types containing the partial type constructor $\mathcal{P}$. The only exception is the field selection, which requires only partial information on types specified by kinds. From an expression e of type of the form $\mathcal{P}([\ldots, l{:}\delta, \ldots])$, the l field can be safely extracted. The result of the field selection $e.l$ is δ itself if δ is a base type. However, if δ is a compound type then the actual type of the l field of the expression e is some δ' such that $\delta \lesssim \delta'$. In this case, the type of the result of field selection $e.l$ is the partial type $\mathcal{P}(\delta)$. Recall the typing rule for field selection:

$$(\text{DOT}) \quad \frac{C, \mathcal{K}, \mathcal{A} \rhd e \ : \ \tau_1 \qquad \mathcal{K} \vdash \tau_1 :: [\![l : \tau_2]\!]}{C, \mathcal{K}, \mathcal{A} \rhd e.l \ : \ \tau_2}$$

To make this rule to be applicable to the above two cases for partial values, we only need to define the following kinding rule for partial types.

$$\mathcal{K} \vdash \mathcal{P}([l_1{:}\delta_1, \ldots, l_n{:}\delta_n, \ldots]) :: [\![l_1{:}\pi_1, \ldots, l_n{:}\pi_n]\!]$$
where $\pi_i = \delta_i$ if δ_i is a base type otherwise $\pi_i = \mathcal{P}(\delta_i)$.

Other rules defined in Figure 5 remain unchanged except that types may contain partial types. A record kind now ranges also over partial types and the field selection becomes polymorphic over partial types as well as complete types.

For this extended language, we still have a complete type inference algorithm. The necessary technique is essentially the same as that for typechecking join operation we have described in the previous section. We then have a language that uniformly integrate heterogeneous sets in its type system. For example, the function

```
Wealthy : {"a::[Name : "b, Salary : int]} -> {"b}
```

```
-> DB;
>> val it = {···} : {any}
-> val employees = filter Employee* DB;
>> val employees = {···} : {P([Name : string, Address : string, Salary : int])}
-> val customers = filter Customer* DB;
>> val customers = {···} : {P([Name : string, Address : string, Balance : int])}
-> union(employees,customers);
>> val it = {···} : {P([Name : string,Address : string])}
-> intersection(employees,customers);
>> val it = {···} : {P([Name : string,Address : string, Balance : int, Salary : int])}
-> fun RichEmployees S = select x.Name from x <- S where x.Salary > 30,000
>> val RichEmployees = fn : {"a::[Salary : int, Name : "b]} -> {"b}
-> fun GoodCustomers S = select x.Name from x <- S where x.Balance > 3,000
>> val GoodSustomers = fn : {"a::[Balance : int, Name : "b]} -> {"b}
-> fun GoodEmployees S = intersection(GoodCustomers(S),RichEmployees(S));
>> val GoodEmployees = fn : {"a::[Balance : int, Salary : int, Name : "b]} -> {"b}
-> GoodEmployees(intersection(employees,customers));
>> val it = {···} : {string}
```

Fig. 18. Programming with Heterogeneous Sets

we defined in the introduction may also be applied to heterogeneous sets of type such as $\{\mathcal{P}([\mathsf{Name} : \mathsf{string}, \mathsf{Salary} : \mathsf{int}])\}$. Figure 18 gives examples involving partial values.

6. CONCLUSIONS

We have demonstrated an extension to the type system of ML which, using *kinded* type inference, allows record formation and field selection to be implemented as polymorphic operations. This together with a set type allows us to represent sets of records – relations – and a number of operations (union, difference, selection and projection onto a single attribute) of a generalized (non first-normal-form) relational algebra. This has been implemented; in particular a recent technique [Ohori 1992; Ohori 1995] for compiling field selection into an efficient indexing operation is being combined with the record operations mentioned above in an extension to Standard ML of New Jersey [Appel and MacQueen 1991].

A further extension to this type system using *conditional* type schemes allows us to provide polymorphic projection and natural join operations, giving a complete implementation of a generalized relational algebra. It could be argued that these operations are not important since they are not present in practical relation query languages. Instead a product and single-column projection are usually employed. However a similar type inference scheme can be used in a technique for statically checking the safety of operations on heterogeneous collections, in which each member of a collection of dynamically typed values have some common structure. The approach we have described provides, we believe, a satisfactory account of how relational database programming, and some aspects of object-oriented programming may be brought into the framework of a polymorphically typed programming language, and it may be used as the basis for a number of further investigations into

the principles of database programming. We briefly review a few here.

Abstract Types and Classes. While we have covered some aspects of object-oriented databases, we have not dealt with the most important aspect of classes in object-oriented programming: that of abstraction and code sharing. In [Ohori and Buneman 1989] statically typed polymorphic class declarations are described. The implementation type of a class is normally a record type, whose fields correspond to "instance variables" in object-oriented terminology. That methods correctly use the implementation type is ensured through checking the correctness of field selection, as described in this paper, and the same techniques may be carried into subclasses to check that code is properly inherited from the superclass. For example, one can define a class Person as:

```
class Person = [Name:string, Age:int]
with
    fun make_person (n,a) = [Name=n, Age=a] : string * int -> Person
    fun name p = p.Name : sub -> string
    fun age p = p.Age : sub -> int
    fun increment_age p = modify(p,Age,p.Age + 1) : sub -> sub
end
```

where sub is a special type variable ranging over the set of all subtypes of Person, which are to be defined later. Inclusion of the sub variable in the type of methods name, age, and increment_age reflects the user's intention being that these methods should be inherited by the subtypes of Person. From this, the extended type system infers the following typing for each method defined in this class.

```
class Person with
    make_person : string * int -> Person
    name : ('a <Person) -> string
    age : ('a <Person) -> int
    increment_age : ('a <Person) -> ('a <Person)
```

The notation ('a <Person) is another form of a kinded type variable whose instances are restricted to the set of subtypes of Person. This can be regarded as an integration of the idea of bounded type abstraction introduced in [Cardelli and Wegner 1985] and data abstraction. As in an object-oriented programming language, one can define a subclasses of Person as:

```
class Employee = [Name:string, Age:int, Salary:int] isa Person
with
    fun make_employee (n,a) = [Name=n, Age=a, Salary=0] : string * int -> Employee
    fun salary e = e.Salary : sub -> int
    fun add_salary (e,s) = modify(e,Salary,e.Salary + s) : sub * int -> sub
end
```

By the declaration of isa Person, this class inherits methods name, age, increment_age from Person. The prototype implementation of Machiavelli prints the following type information for this subclass definition.

```
class Employee isa Person with
    make_employee : string * int -> Employee
    add_salary : ('a <Employee) * int -> ('a <Employee)
    salary : ('a <Employee) -> int
inherited methods:
    name : ('a <Person) -> string
```

```
age : ('a <Person) -> int
increment_age : ('a <Person) -> ('a <Person)
```

The type system can statically check the type consistency of methods that are inherited. It is also possible to define classes that are subclasses of more than one classes, such as ResearchFellow below.

```
class Student = [Name:string, Age:int, Grade:real] isa Person
with
    fun make_student (n,a) = [Name=n, Age=a, Grade=0.0] : string * int -> Employee
    fun grade s = s.Grade : sub -> real
    fun set_grade (s,g) = modify(s,Salary,g) : sub * real -> sub
end

class ResearchFellow = [Name:string, Age:int, Salary:int, Grade:real]
isa {Employee, Student} with
    fun make_RF (n,a) = [Name=n, Age=a, Grade=0.0, Salary = 0]
        : string * int -> ResearchFellow
end
```

Classes can be parameterized by types and the type inference system we have described can be extended to programs involving classes and subclass definitions.

One possible addition to this idea is the treatment of object identity. Throughout this paper we have held to the view that object identity, as a programming construct, is nothing more than reference, and that object creation and update are satisfactorily described by the operations on references given in ML and a number of other programming languages. However Abiteboul and Bonner [Abiteboul and Bonner 1991] have given a catalog of operations on objects and classes, not all of which can be described by means of this simple approach to object identity. Some of the operations appear to call for the passing of reference through an abstraction. For example one may think of Person object identities as references to instances of a Person class and Employee object identities as references to instances of a Employee class. But this approach precludes the possibility that some of the Person and Student identities may be the same, in fact the latter may be a subset of the former. The ability to ask whether two abstractions are both "views" of the same underlying object appears to call for the ability to pass a reference through an abstraction. If this can be done, we believe it is possible to implement most, if not all, the operations suggested by Abiteboul and Bonner.

Other collection types. The original description of Machiavelli [Ohori et al. 1989] attracted some attention [Immerman et al. 1991] because of the use of hom as the basic operation for computation on sets. The reason for using hom was simply to have a small, but adequate collection of operations on sets on which to base our type system. For the purpose of type inference or type checking, the fewer primitive functions the better. In our development, record types and set types are almost independent; there are only a few primitive operations that involve both, and these occur in sections 4 and 5. For other purposes we could equally well have used record types in conjunction with lists, bags or some other collection type. In fact the use of lists, bags and sets is common in object-oriented programming, and some object-oriented databases [Object Design Inc. 1991] supply all three as primitive types.

The study of the commonality between these various collection types is a fruitful extension to the ideas provided here. It may provide us with better ways of structuring syntax [Wadler 1990], with an understanding of the commonality between collection types [Watt and Trinder 1991], and a more general approach to query languages and optimization for these types [Breazu-Tannen et al. 1992].

ACKNOWLEDGMENTS

Val Breazu-Tannen deserves our special thanks. He has contributed to many of the ideas in this paper and has greatly helped us in our understanding of type systems. We thank the referees for their careful reading; we are also grateful for helpful conversations with Serge Abiteboul, Malcolm Atkinson, Luca Cardelli, John Mitchell, Rick Hull and Aaron Watters.

REFERENCES

Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G. 1991. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems 13,* 2, 237–268.

Abiteboul, S. and Bonner, A. 1991. Objects and views. In *Proc. ACM SIGMOD Conference*, pp. 238–247.

Albano, A., Cardelli, L., and Orsini, R. 1985. Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems 10,* 2, 230–260.

Appel, A. W. and MacQueen, D. B. 1991. Standard ML of New Jersey. In *Proc. Third International Symposium on Programming Languages and Logic Programming*, pp. 1–13.

Atkinson, M., Bailey, P., Chisholm, K., Cockshott, W., and Morrison, R. 1983. An approach to persistent programming. *Computer Journal 26,* 4 (November), 360–365.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrick, K., Maier, D., and Zdonik, S. 1989. The object-oriented database system manifesto. In *Proceedings of the First Deductive and Object-Oriented Database Conference*, Kyoto, Japan, pp. 223-240.

Atkinson, M. and Buneman, O.P. 1987. Types and persistence in database programming languages. *ACM Computing Surveys 19,* 1, 105–190.

Augustsson, L. 1984. A compiler for Lazy ML. In *Proc. ACM Symposium on LISP and Functional Programming*, pp. 218–227.

Bancilhon, F., Briggs, T., Khoshafian, S., and Valduriez, P. 1988. FAD, a powerful and simple database language. In *Proc. Intl. Conf. on Very Large Data Bases*, pp. 97–105.

Biskup, J. 1981. A formal approach to null values in database relations. In *Advances in Data Base Theory Vol 1.* New York: Prenum Press.

Breazu-Tannen, V., Buneman, P., and Naqvi, S. 1991. Structural recursion as a query language. In *Proc. 3rd International Workshop on Database Programming Languages*, pp. 9–19. Morgan Kaufmann Publishers.

Breazu-Tannen, V., Buneman, P., and Wong, L. 1992. Naturally embedded query languages. In *Proc. International Conference on Database Theory, Springer LNCS,* pp. 140-154.

Breazu-Tannen, V. and Subrahmanyam, R. 1991. Logical and computational aspects of programming with sets/bags/lists. In *Proc. International Colloquium on Automata, Languages, and Programming, Springer LNCS 510*, pp. 60–75.

Buneman, P., Jung, A., and Ohori, A. 1991. Using powerdomains to generalize relational databases. *Theoretical Computer Science 91,* 1, 23–56.

Buneman, P., Libkin, L., Suciu, D., and Tannen, V.and Wong, L. 1994. Comprehension syntax. *SIGMOD Record 23,* 1, 87–96.

Buneman, P. and Ohori, A. 1991. A type system that reconcile classes and extents. In *3rd International Workshop on Database Programming Languages* pp. 191–202. Morgan Kaufmann Publishers.

Cardelli, L. 1986. Amber. In *Combinators and Functional Programming, Lecture Notes in Computer Science 242*, pp. 21–47. Springer-Verlag.

Cardelli, L. 1988. A semantics of multiple inheritance. *Information and Computation 76*, 138–164. (Special issue devoted to Symp. on Semantics of Data Types, Sophia-Antipolis, France, 1984).

Cardelli, L. and Mitchell, J. 1989. Operations on records. In *Proceedings of Mathematical Foundation of Programming Semantics, Lecture Notes in Computer Science 442*, pp. 22–52.

Cardelli, L. and Wegner, P. 1985. On understanding types, data abstraction, and polymorphism. *Computing Surveys 17,* 4 (Dec.), 471–522.

Copeland, G. and Maier, D. 1984. Making Smalltalk a database system. In *Proc. ACM SIGMOD conference*, pp. 316–325.

Courcelle, B. 1983. Fundamental properties of infinite trees. *Theoretical Computer Science 25*, 95–169.

Damas, L. and Milner, R. 1982. Principal type-schemes for functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 207–212.

Gallier, J. and Snyder, W. 1989. Complete sets of transformations for general E-unification. *Theoretical Computer Science 67,* 2, 203–260.

Harper, R. and Pierce, B. 1991. A record calculus based on symmetric concatenation. In *Proc. ACM Symposium on Principles of Programming Languages,* pp. 131–142.

Hart, H. and Wong, L. 1994. Query language for genetic databases. Unpublished manuscript. Available on WWW via `http://www.cis.upenn.edu/~wfan/DBHOME.html`.

Hindley, R. 1969. The principal type-scheme of an object in combinatory logic. *Trans. American Mathematical Society 146*, 29–60.

Hoang, M., Mitchell, J., and Viswanathan, R. 1993. Standard ML weak polymorphism and imperative constructs. In *Proc. IEEE Symposium on Logic in Computer Science,* pp. 15–25.

Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., and Perterson, J. 1992. Report on programming language Haskell a non-strict, purely functional language version 1.2. *SIGPLAN Notices, Haskell special issue 27,* 5.

Huet, G. 1976. Résolution d'équations dans les langages d'ordre 1,2,... ω. Ph. D. thesis, University Paris.

Ichbiah, J., Barnes, J., Heliard, J., Krieg-Bruckner, B., Roubine, O., and Wichmann, B. 1979. Rationale of the design of the programming language Ada. *ACM SIGPLAN notices 14,* 6.

Imielinski, T. and Lipski, W. 1984. Incomplete information in relational databases. *Journal of ACM 31,* 4 (Oct.), 761–791.

Immerman, N., Patnaik, S., and Stemple, D. 1991. The expressiveness of a family of finite set languages. In *Proc. ACM Symposium on Principles of Database Systems*, pp. 37–52.

Jategaonkar, L. A. and Mitchell, J. 1988. ML with extended pattern matching and subtypes. In *Proc. ACM Conference on LISP and Functional Programming*, Snowbird, Utah, pp. 198–211.

Kim, W. 1994. Observations on the ODMG-93 proposal. *ACM Sigmod record 23,* 1.

Leroy, X. 1993. Polymorphism by names for references and continuation. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 220–231.

Leroy, X. and Weise, P. 1991. Polymorphic type inference and assignment. In *Proc. ACM Symposium on Principles of Programming Languages,* 291–302.

Lipski, W. 1979. On semantic issues connected with incomplete information databases. *ACM Transactions on Database Systems 4,* 3 (Sept.), 262–296.

MacQueen, D. 1988. References and weak polymoprhism. Note in Standard ML of New Jersey Distribution Package.

Milner, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 348–375.

Milner, R., Tofte, M., and Harper, R. 1990. *The Definition of Standard ML.* The MIT Press.

Mitchell, J. 1990. Type systems for programming languages. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Chapter 8, pp. 365–458. MIT Press/Elsevier.

Morrison, R., Brown, A., Connor, R., and Dearle, A. 1989. Napier88 reference manual. Tech. rep., Department of Computational Science, University of St Andrews.

Object Design Inc. 1991. *ObjectStore Reference Manual.* Burlington, MA. Object Design Inc.

Ohori, A. 1989a. A simple semantics for ML polymorphism. In *Proc. ACM/IFIP Conference on Functional Programming Languages and Computer Architecture*, London, England, pp. 281–292.

Ohori, A. 1989b. A study of types, semantics and languages for databases and object-oriented programming. Ph. D. thesis, University of Pennsylvania.

Ohori, A. 1990. Semantics of types for database objects. *Theoretical Computer Science 76*, 53–91.

Ohori, A. 1992. A compilation method for ML-style polymorphic record calculi. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 154–165.

Ohori, A. 1995. A polymorphic record calculus and its compilation. Submitted for publication. Available as a preprint RIMS-1013 from RIMS, Kyoto University. (Extended version of [Ohori 1992].)

Ohori, A. and Buneman, P. 1988. Type inference in a database programming language. In *Proc. ACM Conference on LISP and Functional Programming*, Snowbird, Utah, pp. 174–183.

Ohori, A. and Buneman, P. 1989. Static type inference for parametric classes. In *Proc. ACM OOPSLA Conference*, New Orleans, Louisiana, pp. 445–456. (The extended version was published in Mitchell, J. and Gunter, G. editors, *Theoretical Aspects of Object-Oriented Programming,* pp. 121–147, 1994. MIT Press.)

Ohori, A., Buneman, P., and Breazu-Tannen, V. 1989. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proc. ACM SIGMOD conference*, Portland, Oregon, pp. 46–57.

Remy, D. 1989. Typechecking records and variants in a natural extension of ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 77–88. (The extended version was published in Mitchell, J. and Gunter, G. editors, *Theoretical Aspects of Object-Oriented Programming,* pp. 67–95, 1994. MIT Press.)

Remy, D. 1992. Typing record concatenation for free. In *Proc. ACM Symposium on Principles of Programming Languages,* pp. 166–175.

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of ACM 12*, 23–41.

Schmidt, J. 1977. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems 5,* 2.

Stonebraker, M. and Rowe, L. 1986. The design of Postgres. In *Proc. ACM SIGMOD conference*, pp. 340–355.

Stroustrup, B. 1987. *The C++ programming language.* Addison-Wesley.

Tofte, M. 1988. Operational semantics and polymorphic type inference. Ph. D. thesis, Department of Computer Science, University of Edinburgh.

Turner, D. 1985. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201*, pp. 1–16. Springer-Verlag.

Wadler, P. 1990. Comprehending monads. In *Proc. ACM Conference on Lisp and Functional Programming,* pp. 61–78.

Wand, M. 1987. Complete type inference for simple objects. In *Proc. IEEE Symposium on Logic in Computer Science*, Ithaca, New York, pp. 37–44.

Wand, M. 1988. Corrigendum : Complete type inference for simple object. In *Proc. IEEE Symposium on Logic in Computer Science,* pp. 132.

Wand, M. 1989. Type inference for record concatenation and simple objects. In *Proc. IEEE Symposium on Logic in Computer Science*, pp. 92–97.

Watt, D. and Trinder, P. 1991. Towards a theory of bulk types. Tech. rep., Department of Computing Science, Glasgow University, Glasgow G12 8QQ, Scotland.

Wirth, N. 1977. Modula: a language for modular multiprogramming. *Software Practice and Experience 7,* 1, 3–35.

Wong, L. 1994. Querying nested collections. Ph. D. thesis, Department of Computer and Information Science, University of Pennsylvania.

Zaniolo, C. 1984. Database relation with null values. *J. Comput. Syst. Sci. 28,* 1, 142–166.